# PyTorch Artificial Intelligence Fundamentals

A recipe-based approach to design, build and deploy your own AI models with PyTorch 1.x

Jibin Mathew

Packt
www.packt.com

VISIT…

# PyTorch Artificial Intelligence Fundamentals

A recipe-based approach to design, build and deploy your own AI models with PyTorch 1.x

**Jibin Mathew**

Packt

**BIRMINGHAM - MUMBAI**

# PyTorch Artificial Intelligence Fundamentals



**Commissioning Editor:** Amey Varangoankar
**Acquisition Editor:** Reshma Raman
**Content Development Editor:** Nathanya Dias
**Senior Editor:** Ayaan Hoda
**Technical Editor:** Joseph Sunil
**Copy Editor:** Safis Editing
**Project Coordinator:** Aishwarya Mohan
**Proofreader:** Safis Editing
**Indexer:** Tejal Daruwale Soni
**Production Designer:** Deepika Naik

First published: February 2020

Production reference: 1270220

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-83855-704-1

www.packt.com

*"I can do all things through Christ who strengthens me."*

*- Philippians 4:13*

# Contributors

## About the author

**Jibin Mathew** is a senior data scientist and machine learning researcher who has worked in the AI domain for more than 7 years. He is a serial entrepreneur and has founded multiple AI start-ups. He has a strong software engineering background and understands the complete workflow, from research to scalable production deployment. He has built solutions in the fields of healthcare, environment, finance, industrial monitoring, and retail. He has been an adviser to various companies in their AI endeavors. He was the winner of Singularity University's Global Impact Challenge 2018 and has been part of various global platforms. He is an active contributor to the community and shares his knowledge by authoring content and through blog posts.

# About the reviewers

**Ganesh Naik** is an author, consultant and corporate trainer in the artificial intelligence, data science, machine learning, and embedded Linux product development fields. He is a graduate in computer engineering and has industry experience. He has authored books such as *Mastering Python Scripting for System Administrators* and *Learning Linux Shell Scripting* and is the coauthor of *Bash Cookbook*, Packt Publishing. Ganesh has a passion for teaching. He has trained thousands of engineers in AI, ML, and Linux product development. He has worked as a corporate trainer for corporates such as the Indian Space Research Organisation, Intel, GE, Samsung, Motorola, and various other corporates in India and various other countries.

**Rafal Pronko** has been a data scientist for 6 years. In his professional life, he has worked on projects such as ad categorization, extracting information from short descriptions; object detection (Smart Shelf and CleanAI projects); and recognition, face, and anti-spoofing recognition mechanisms (the SmartBar anti-spoofing app). Now, he is working for CV Timeline on extracting information from unstructured text such as resumes and social media profiles, categorizing the data, and helping recruiters make better decisions.

# Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

# Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

# Table of Contents

# Preface

**Artificial Intelligence** (**AI**) continues to grow in popularity and disrupt a wide range of domains, but it is a complex and daunting topic. In this book, you'll get to grips with building deep learning apps, and how you can use PyTorch for research and solving real-world problems.

This book uses a recipe-based approach, starting with the basics of tensor manipulation, before covering **Convolutional Neural Networks** (**CNNs**) and **Recurrent Neural Networks** (**RNNs**) in PyTorch. Once you are well-versed with these basic networks, you'll build a medical image classifier using deep learning. Next, you'll use TensorBoard for visualizations. You'll also delve into **Generative Adversarial Networks** (**GANs**) and **Deep Reinforcement Learning** (**DRL**) before finally deploying your models to production at scale. You'll discover solutions to common problems faced in machine learning, deep learning, and reinforcement learning. You'll learn to implement AI tasks and tackle real-world problems in computer vision, **natural language processing** (**NLP**), and other real-world domains.

By the end of this book, you'll have the foundations of the most important and widely used techniques in AI using the PyTorch framework.

## Who this book is for

This PyTorch book is for AI engineers who are just getting started and machine learning engineers, data scientists, and deep learning enthusiasts who are looking for a guide to help them solve AI problems effectively. Working knowledge of the Python programming language and a basic understanding of machine learning are expected.

## What this book covers

`Chapter 1`, *Working with Tensors Using PyTorch*, introduces PyTorch and its installation and then jumps on to working with tensors using PyTorch.

`Chapter 2`, *Dealing with Neural Networks*, goes through all of the requirements to get started and train a fully connected neural network, providing a thorough explanation of all the components of a basic neural network: layers, feedforward network, backpropagation, loss functions, gradients, weight updates, and using a CPU/GPU.

`Chapter 3`, *Convolutional Neural Networks for Computer Vision*, starts by looking at a class of neural networks for more advanced tasks, called convolutional neural networks. Here, we will explore TorchVision alongside PyTorch, train a CNN model, and visualize its progress using TensorBoard. We will also cover various tasks related to the building blocks of convolutional networks. A convolutional neural network (CNN or ConvNet) is a class of DNN that is most commonly applied to analyze images.

`Chapter 4`, *Recurrent Neural Networks for NLP*, explores recurrent neural networks and looks at various modifications within RNNs, as well as best practices.

`Chapter 5`, *Transfer Learning and TensorBoard*, shows how to train an image classifier to distinguish normal and pneumonia chest X-rays, using a trained ResNet-50 model to perform transfer learning. We will replace the classifier and have two output units to represent the Normal and Pneumonia classes.

`Chapter 6`, *Exploring Generative Adversarial Networks*, explores generative adversarial networks and how to implement the components of PyTorch and train an end-to-end network. We will explore DCGANs and further improve the limitations of DCGANs with a progressive GAN network.

`Chapter 7`, *Deep Reinforcement Learning*, helps you to gain an understanding of deep RL with various recipes. This chapter is a series of recipes and tasks where you'll utilize the abilities and architectures you need to turn into a deep reinforcement learning expert.

`Chapter 8`, *Productizing AI models in PyTorch*, looks at productizing PyTorch applications in two ways. Firstly, productizing an already trained model, and secondly, performing distributed training on large datasets. Finally, we will look at portability between various frameworks.

# To get the most out of this book

Working knowledge of Python is required.

# Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.

2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/PyTorch-Artificial-Intelligence-Fundamentals`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

# Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `http://www.packtpub.com/sites/default/files/downloads/9781838557041_ColorImages.pdf`.

# Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "For Linux, we will use the following `pip` manager."

A block of code is set as follows:

```
a = np.ones((2, 3))
a
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
b.shape
torch.Size([2, 3])
```

Any command-line input or output is written as follows:

```
pip3 install
https://download.pytorch.org/whl/cu90/torch-1.1.0-cp36-cp36m-win_amd64.whl
```

**Bold**: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "A **scalar** is a single independent value."

Warnings or important notes appear like this.


Tips and tricks appear like this.

# Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

## Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

## How to do it…

This section contains the steps required to follow the recipe.

## How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

# There's more…

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

# See also

This section provides helpful links to other useful information for the recipe.

# Get in touch

Feedback from our readers is always welcome.

**General feedback**: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

**Errata**: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

**Piracy**: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

**If you are interested in becoming an author**: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

# Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

# 1
# Working with Tensors Using PyTorch

Deep learning is a subfield within the parent field of machine learning, which is the study and application of a class of algorithms inspired by the working of the brain. Given enough data and iteration through it, these algorithms can approximate any function that describes the data, and are rightly called universal function approximators. So where does PyTorch come into this ecosystem?

PyTorch is an open-source deep learning framework in Python that lets us start by examining a problem, come up with a prototype solution, and progress in our development of this solution all the way up to the creation of a distributed compute cluster. It keeps you covered from research to production. PyTorch is adapted from Torch, which is a scientific computing framework with wide support for machine learning algorithms that provides you with a great ability (using GPUs) and is written in Lua. So why PyTorch?

PyTorch is deeply integrated with Python, has an imperative style, uses a Python-like syntax, and is easy to use and flexible in Eager mode. It has a very shallow learning curve and lets you focus on the functionality rather than the boilerplate and syntax of the framework. A pure imperative execution of Python code would miss a lot of optimization opportunities, and so, with the introduction of **just-in-time** (**JIT**) compilers, PyTorch allows a transition to graph mode for speed, functionality, and optimization in C++ runtime environments. It has great community support from professionals from different domains, and plays well with libraries. It has native **Open Neural Network Exchange** (**ONNX**) support for interoperability with frameworks. It is distributed, scales to production, integrates with TensorBoard, and has great documentation and APIs, and you can easily write custom extensions for CPUs and GPUs. We will explore these and more in upcoming chapters.

In this chapter, we will cover the following recipes:

- Installing PyTorch
- Creating tensors in PyTorch
- Interoperating NumPy bridge
- Gradients and no gradients
- Viewing tensors in PyTorch

# Technical requirements

To work through this chapter, you need to have Python3 installed. You will also require any modern machine, but you don't need a GPU-enabled device for this chapter. If you want to leverage GPU capabilities, you can use NVIDIA CUDA-enabled GPUs.

# Installing PyTorch

We will install PyTorch in this section.

NumPy is an essential library for this chapter and will be automatically installed for you while you install PyTorch as part of its dependency. This means that we need not explicitly install NumPy.


You can install PyTorch with other package managers, such as conda, as described at `https://pytorch.org/`.

To install PyTorch for Python3 CPU, we can use the following commands:

- For Linux, we will use the following `pip` manager:

```
pip3 install torch==1.4.0+cpu -f
https://download.pytorch.org/whl/torch_stable.html
```

- For Windows, we will use the following `pip` manager:

```
pip3 install torch==1.4.0+cpu -f
https://download.pytorch.org/whl/torch_stable.html
```

- For MacOS, we will use the following `pip` manager:

```
pip3 install torch
```

To install PyTorch for tbe Python3 CUDA-enabled GPU version, we can use the following:

- For Linux, we will use the following `pip` manager:

```
pip3 install torch
```

- For Windows, we will use the following `pip` manager:

```
pip3 install
https://download.pytorch.org/whl/cu90/torch-1.1.0-cp36-cp36m-win_amd64.whl
```

MacOS Binaries don't support CUDA, so you should install it from the source if you need CUDA. You can alternatively install it using other package managers and even build it from the source. For other package managers and Python versions, visit `https://pytorch.org/`.

You can quickly verify the installation works OK by going to the Python terminal and typing in the following commands:

```
import torch
import numpy
```

If these imports worked fine, you are good to go!

# Creating tensors in PyTorch

Let's first understand what a tensor is. A **scalar** is a single independent value, a 1D array of values is called a **vector**, a 2D array of values is called a **matrix**, and any array of values that is more than 2D is simply called a **tensor**. A tensor is a generalized term that encompasses scalars, vectors, and matrices.

A scalar is a $0^{th}$ order tensor, a vector is a $1^{st}$ order tensor and a matrix is a $2^{nd}$ order tensor.

The following are the various tensors:

- **Scalar**: This is a zeroth-order tensor. An example of a scalar is $x1$.
- **Vector**: This is a first-order tensor; the following is an example of a vector:

$$[x1 \quad x2 \quad x3] \, or \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix}$$

- **Matrix**: This is a second-order tensor. The following is an example of a matrix:

$$\begin{bmatrix} x1 & x2 \\ x3 & x4 \end{bmatrix}$$

- **Tensors:** These are anything above a second-order tensor, as shown by the following example:

$$\begin{bmatrix} \begin{bmatrix} x1 & x2 \\ x3 & x4 \end{bmatrix} & \begin{bmatrix} x1 & x2 \\ x3 & x4 \end{bmatrix} \\ \begin{bmatrix} x1 & x2 \\ x3 & x4 \end{bmatrix} & \begin{bmatrix} x1 & x2 \\ x3 & x4 \end{bmatrix} \end{bmatrix}$$

With this, we will move on to our recipe showing how to work with tensors.

# How to do it...

There are multiple ways to create a tensor in PyTorch. We will look at a few of them in this section:

- We can create a tensor with all ones as follows:

    1. Let's start by importing the library:

    ```
    import torch
    ```

    2. We will use the `ones()` method:

    ```
    torch.ones((2,3))
    ```

This will return a tensor that contains ones and has a default float datatype as follows:

```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

- Now, we will create a tensor consisting of only integer ones:

    1. We will do exactly the same as in the previous recipe, but we will add datatype (`dtype`) as a parameter:

    ```
    torch.ones((2,3), dtype=torch.int8)
    ```

    2. This will return a tensor consisting only of integer ones:

    ```
    tensor([[1, 1, 1],
            [1, 1, 1]], dtype=torch.int8)
    ```

- Next, we will create a tensor consisting of only integer zeros:

    1. We will do exactly the same as before, but using the `zeros()` method:

    ```
    torch.zeros((2,3), dtype=torch.int8)
    ```

    2. This will return a tensor consisting of only integer zeros:

    ```
    tensor([[0, 0, 0],
            [0, 0, 0]], dtype=torch.int8)
    ```

- We will now create a tensor filled with a specific value:

    1. We will use the `full()` method and pass in the required fill value along with the shape:

    ```
    torch.full((2, 3), 3.141592)
    ```

    2. This will return a tensor with the given value:

    ```
    tensor([[3.1416, 3.1416, 3.1416],
            [3.1416, 3.1416, 3.1416]])
    ```

Note that the values are rounded off.

- Now, we will create an empty tensor:

    1. We will use the `empty()` method for this:

    ```
    torch.empty((2,3))
    ```

    2. This will return a tensor filled with uninitialized data, varying each time and for each machine:

    ```
    tensor([[2.5620e-01, 4.5773e-41, 2.5620e-01],
            [4.5773e-41, 4.4842e-44, 0.0000e+00]])
    ```

- We will next create a tensor from a uniform distribution:

    1. We will use the `rand()` method:

    ```
    torch.rand((2,3))
    ```

    2. This will draw a tensor with random values from a uniform distribution from [0, 1]:

    ```
    tensor([[0.6714, 0.0930, 0.4395],
            [0.5943, 0.6582, 0.6573]])
    ```

- We will create a tensor with mean 0 and variance 1:

    1. We will use the `randn()` method:

    ```
    torch.randn((2,3))
    ```

    2. This will draw a tensor with random values with mean 0 and variance 1 from a normal distribution, also called the **standard normal distribution**:

    ```
    tensor([[ 0.3470, -0.4741, 1.2870],
            [ 0.8544, 0.9717, -0.2017]])
    ```

- Next, we will create a tensor from a given range of values

    1. We will use the `rand_int()` method, passing in the lower limit, the upper limit, and the shape:

    ```
    torch.randint(10, 100, (2,3))
    ```

    2. This will return a tensor between 10 and 100, similar to the following:

    ```
    tensor([[63, 93, 68],
            [93, 58, 29]])
    ```

- We will now create a tensor from existing data:

    1. We will use the `tensor` class for this:

    ```
    torch.tensor([[1, 2 ,3], [4, 5, 6]])
    ```

    This will create a copy of the data and create a tensor. If you want to avoid making a copy, you could use `torch.as_tensor([[1, 2 ,3], [4, 5, 6]])`.

    2. This returns a tensor with the same datatype as that of the data, which in this case is an integer tensor:

    ```
    tensor([[1, 2, 3],
            [4, 5, 6]])
    ```

    Also, note that, if one of the data values is a float, then all of the values would be converted into a float; however, if one of the values is a string, then an error is thrown.

- Next we will create a tensor with the attributes from another tensor:

    1. Let's first create a reference tensor for this:

    ```
    a = torch.tensor([[1, 2 ,3], [4, 5, 6]])
    ```

    2. Let's see the datatype of tensor `a`:

    ```
    a.dtype
    torch.int64
    ```

    3. Now let's look at the shape of the tensor:

    ```
    a.shape
    torch.Size([2, 3])
    ```

    4. The datatype and shape meet our expectation, so now let's create a tensor `b` so that it matches the attributes of `a` and use the `torch.*_like` format for this:

    ```
    b = torch.ones_like(a)
    b
    ```

This results in the following output:

```
tensor([[1, 1, 1],
        [1, 1, 1]])
```

5. Let's see the datatype of tensor `b`:

```
b.dtype
torch.int64
```

6. Let's also look at the shape of tensor `b`:

```
b.shape
torch.Size([2, 3])
```

- Next, we will create a tensor with a similar type to another tensor, but of a different size:

1. We will use the same tensor `a`, from the previous step and use the `torch.new_*` format for this:

```
a.new_full((2,2), 3.)
```

2. This returns the following output:

```
tensor([[3, 3],
        [3, 3]])
```

These are the different methods to create tensors in PyTorch.

# How it works...

In this recipe, we had a look at the various methods for creating tensors from various data sources. Before we start exploring the concept of deep learning with PyTorch and how it works, it is essential to understand some of the most commonly used functionalities for dealing with the basic unit of data, tensors. We can use the `torch.tensor()` method to create tensors with various kinds of values and shapes. We could even draw a tensor from a uniform distribution or standard normal distribution, which are essential in initializing a neural network for optimal performance and training time, and all these tensors have a default `torch.FloatTensor` datatype and update the datatype using the `dtype` parameter.

The `.ones()` method creates a tensor containing 1 in the given shape, `.zeros()` fills the tensors with all zeros, and the `full()` method fills the tensors with the given value. The `.empty()` method creates an empty tensor, `.rand()` draws a tensor with random values from a uniform distribution from [0, 1), and `.randn()` draws a tensor with random values with mean 0 and variance 1 from a normal distribution, also called the standard normal distribution.

The `rand_int()` method draws random integers from a given range and creates a tensor in a given shape. We can create tensors with the shape of another tensor, a tensor with all ones, but the shapes and datatype of another tensor can be created using the `ones_like()` method. We can use the `torch.new_*` format to create a tensor with a similar type to another tensor, but a different size.

We can also take data from an existing source and convert it into tensors, and there are advanced tensor creation techniques that reduce the memory footprint and use the shape of an existing tensor and/or the datatype of a tensor.

## There's more...

You can find the shape of a tensor using the `shape` attribute or `size()` method and the datatype using the `dtype` attributes of a tensor. You can also use `torch.numel()` to get the total number of elements in a tensor.

## See also

To learn more, read PyTorch's official documentation for tensor creation options at `https://pytorch.org/docs/stable/tensors.html#torch.Tensor`.

# Exploring the NumPy bridge

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object and various derived objects. Beyond this, NumPy is used as an efficient container for generic multidimensional data. NumPy allows for seamless and speedy integration with a wide variety of databases.

NumPy is the standard Python library and is used to deal with numerical data. Many well-known ML/DS libraries in Python, such as pandas (a library that is used to read data from many sources) and scikit-learn (one of the most important ML libraries, used to read and write images) use NumPy under the hood. You will deal with `numpy` a lot, for example, while dealing with tabular data, loading it using the `pandas` library and getting `numpy` arrays out of the dataframe; reading images, where many existing libraries have in-built APIs for reading them as `numpy` arrays; and also converting `numpy` arrays into images, as well as text and other forms of data. Also, these all support `numpy` arrays using `scikit-learn`, which is a machine learning library. As you can see, it is important to have a bridge between `numpy` arrays and PyTorch tensors.

# How to do it...

Let's start by importing `numpy`:

1. We will start by creating a `numpy` array; for this, let's import `numpy`:

```
import numpy as np
```

2. We will create a `numpy` array consisting of only `ones`:

```
a = np.ones((2, 3))
a
```

This results in the following output:

```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

3. Now we will convert it into a PyTorch tensor:

```
b = torch.from_numpy(a)
b
```

This results in the following output:

```
tensor([[1., 1., 1.],
        [1., 1., 1.]], dtype=torch.float64)
```

4. Now we will convert a tensor to a `numpy` array:

```
b.numpy()
```

This results in the following output:

```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

With this recipe, we've now got the hang of moving back and forth between NumPy and Torch tensors.

# How it works...

We started by importing `numpy` to create a `numpy` array. Then, we created a `numpy` array consisting of only ones using `np.ones()`, which converted it into a PyTorch tensor using the `from_numpy()` method. Then we converted the tensor to a `numpy` array using the `.numpy()` method.

It is extremely easy to switch between a PyTorch tensor and NumPy; in fact, it can be achieved with just two methods. This makes it possible to take a predicted tensor and convert into an image from NumPy (using a library that supports NumPy-to-image conversion) and similarly back from NumPy to a tensor.

# There's more...

The underlying memory is shared between a NumPy array and a PyTorch tensor, and hence any change made in one would affect the other.

Let's have a look at how this is presented in the following code block:

```
>>a
array([[1., 1., 1.],
       [1., 1., 1.]])

>>b = torch.from_numpy(a)
>>b
tensor([[1., 1., 1.],
        [1., 1., 1.]], dtype=torch.float64)

>>a*=2
>>a
array([[2., 2., 2.],
```

```
        [2., 2., 2.]])

>>b
tensor([[2., 2., 2.],
        [2., 2., 2.]], dtype=torch.float64)
```

We can see that the changes from the `numpy` are reflected in the tensor as well.

# See also

To learn more, click on PyTorch's official documentation link for the NumPy bridge at `https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#numpy-bridge`.

# Exploring gradients

Let's briefly go through what gradients are. For this, we need to first understand what a gradient descent is. In machine learning problems, we provide an input and desired output pair and ask our model to generalize the relationship between the given input and output pair. But sometimes the model learns that its predictions would be way off from the desired output (this difference is known as a **loss**). So what is gradient descent?

**Gradient descent** is an optimization algorithm used to minimize a function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. We use it while training a model so that it minimizes the loss. It is used to find the values of a function's parameters (coefficients or weights in machine learning) that minimize the cost or loss function.

So what is a gradient? A gradient measures how much the output of the given function varies when varying the inputs by a small factor, which is the same as the concept of derivatives in calculus. A gradient calculates the variation in all weights with respect to the change in error. Gradients are the slope of a function. A higher gradient means a steeper slope and that a model can learn more rapidly. The gradient points toward the direction of steepest slope. The `Autograd` module in PyTorch performs all gradient calculations in PyTorch. It is the core Torch package for automatic differentiation. It uses a tape-based system for automatic differentiation. In the forward phase, the `Autograd` tape will remember all the operations it executed, and in the backward phase it will replay them.

# How to do it...

Let's start by creating a tensor.

1. Unlike the tensors that we have created so far, we will add a new key that lets PyTorch know that it needs to perform gradient calculations on the following tensor:

```
x = torch.full((2,3), 4, requires_grad=True)
x
```

This results in the following output:

```
tensor([[4., 4., 4.],
        [4., 4., 4.]], requires_grad=True)
```

2. Let's create another tensor, `y`, that is derived out of tensor `a`; we will see the difference in the output of this new tensor, as it has a gradient function attached to it:

```
y = 2*x+3
y
```

This results in the following output:

```
tensor([[11., 11., 11.],
        [11., 11., 11.]], grad_fn=<AddBackward0>)
```

3. Let's further explore gradients in PyTorch, starting with the original `x`:

```
x
```

This results in the following output:

```
tensor([[4., 4., 4.],
        [4., 4., 4.]], requires_grad=True)
```

4. We will then define `y`, a slightly more complex tensor than the previous example:

```
y = (2*x**2+3)
y
```

This results in the following output:

```
tensor([[35., 35., 35.],
        [35., 35., 35.]], grad_fn=<AddBackward0>)
```

5. Next, we will calculate gradients with respect to `x` on `y`, since `y` is a tensor, and we want to calculate the gradient with respect to this tensor. To do this, we will pass the shape of `x`, which is the same as `y`:

```
y.backward(torch.ones_like(x))
```

6. Now, let's see the value of the gradient of `x` using the `grad` attribute:

```
x.grad
```

This results in the following output:

```
tensor([[16., 16., 16.],
        [16., 16., 16.]])
```

7. Moving on to the no-gradient part of this section, we can turn off the gradient calculation at a certain point in the code by going through the following steps, first using the `requires_grad_()` method on the tensor, if we revisit tensor `x`:

```
>> x.requires_grad

True

>> x.requires_grad_(False) # turning of gradient
>> x.requires_grad

False
```

8. We can turn off tracking the gradient calculation by using the `.no_grad()` method, starting with `x`:

```
>> x = torch.full((2,3), 4,requires_grad=True)
>> x

tensor([[4., 4., 4.],
        [4., 4., 4.]], requires_grad=True)

>> x.requires_grad

True

>> with torch.no_grad():
..         print((x**5+3).requires_grad)

False
```

With this, we have explored some functionalities of the `Autograd` package.

# How it works...

We can see that Autograd keeps track of operations; when we create the tensor `y` from `x`, `y=2*x+3`, we see that a gradient function, `grad_fn`, is attached to the tensor.

We started by creating a new type of tensor that has `require_grad` set to `True`, after which we created a tensor `y`, such that, $y = 2x^2 + 3$ and discovered that `y` has a different gradient function attached to it. We also looked at using `requires_grad_()`, and finally `no_grad()`.

PyTorch has a package called `autograd` that performs all the tracking and automatic differentiation for all operations on tensors. It is a define-by-run framework, which means that your backpropagation is defined by how your code is run and that every single iteration can be different. We utilized the `require_grad` attribute of the `torch.Tensor` class to determine the state of the gradient calculation and, upon calling the `.backward()` method, automatically computed all of the gradients and the gradient of the tensor in its `.grad` attribute.

We can disable the gradient calculation between the code and also temporarily disable the tracking of tensors for the gradient calculation, increasing the speed of computation. This disabling of the calculation is mostly used during evaluation.

# There's more...

You can use the `torch.set_grad_enabled()` method to enable and disable gradient calculation and the `detach()` method for the future tracking of computations. Use the `grad_fn` attribute to see the gradient function attached to the tensor.

# See also

To learn more, you can check the official documentation at `https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html` and `https://pytorch.org/docs/stable/autograd.html`.

# Viewing tensors in PyTorch

While working with tensors and dealing with neural networks, we often need to go through and rearrange data in the tensors so that the dimensions of the tensors fit the needs of the architecture. In this section, we will explore common rearrangement and reshaping techniques in PyTorch.

In this recipe, we will learn about getting a tensor to look the way we want.

# How to do it...

Let's look at how to change the shape of tensors:

1. First, we will create a tensor, `a`:

```
>>a = torch.Tensor([1, 2, 3, 4])
```

2. We will then use the `reshape()` method:

```
>>torch.reshape(a, (2, 2))
```

This results in the following output:

```
tensor([[1., 2.],
        [3., 4.]])
```

3. Next, we will look at the `resize_()` method:

```
>>a = torch.Tensor([1, 2, 3, 4, 5, 6])
>>a.shape
torch.Size([6])
>>a.resize_((2, 2))
```

This results in the following output:

```
tensor([[1., 2.],
        [3., 4.]])
```

4. The most common method is `view()`:

```
>>a = torch.Tensor([1, 2, 3, 4, 5, 6])
>>a.view((2, 3))
```

This results in the following output:

```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

5. With the `view()` method, you can choose not to mention one of the dimensions and arrange the rest of them, and PyTorch will calculate the missing dimension as follows:

```
>>a.view((2, -1))
```

This results in the following output:

```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

These are the different ways to reshape tensors.

# How it works...

In the previous recipe, we manipulated tensors to change their shape based on the network architecture, looking at three different methods, each applying to a different use case:

- **The `.reshape()` method**: `.reshape(a, b)` returns a new tensor with the same data as the original tensor with size `(a, b)` as it copies the data to another part of memory; `.reshape()` can operate on both contiguous and noncontiguous tensors, and may return a copy or a view of the original tensor.
- **The `.resize()` method**: `.resize_(a, b)` returns the same tensor without creating a copy with the new given shape. But we should keep in mind that, if the new shape results in fewer elements than the original tensor, then it won't throw any error, and some elements will be removed from the tensor but not from memory. If the new shape results in more elements than the original tensor, new elements will be uninitialized in memory without throwing any error.
- **The `.view()` method**: `.view(a, b)` will return a new tensor with the same data as weights with size (a, b); `.view()` can only operate on a contiguous tensor and returns the same storage as the input.

# There's more...

You can use the dimension of another tensor and make a given tensor resemble the dimension of that tensor without affecting the actual dimensions of either of them.

Look at the following code block:

```
>>a = torch.Tensor([[1, 2, 3],
                    [4, 5, 6]])
>>a

tensor([[1., 2., 3.],
        [4., 5., 6.]])

>>b = torch.Tensor([4,5,6,7,8,9])
>>b
tensor([4., 5., 6., 7., 8., 9.])
>>b.view_as(a)

tensor([[4., 5., 6.],
        [7., 8., 9.]])
```

From this, we can see that the tensor `b` takes the shape of tensor `a`.

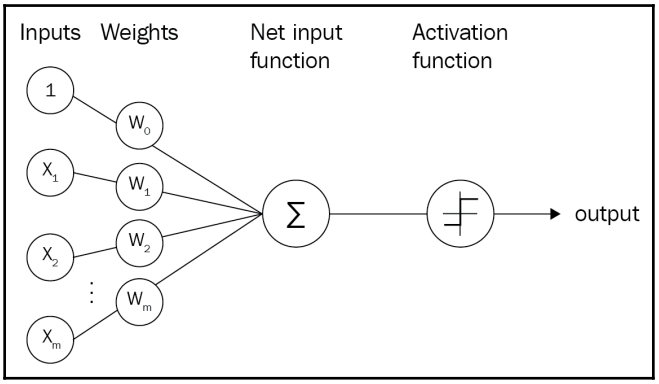
# See also

For more information, you can look at the documentation at `https://pytorch.org/docs/stable/tensors.html#torch.Tensor.view` and `https://pytorch.org/docs/stable/torch.html#torch.reshape`.

# 2
# Dealing with Neural Networks

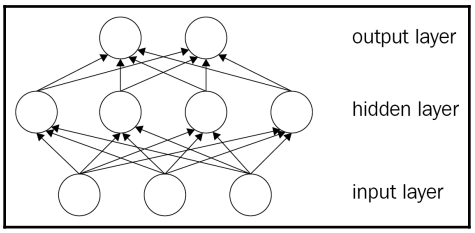
Deep learning is a class of machine learning algorithms that is designed to loosely mimic the neurons in our brain. A neuron takes an input from a number of inputs from surrounding neurons and sums it up, and if the sum exceeds a certain threshold, then the neuron fires. Between each neuron there is a gap called a synapse. Signals are carried across these synapses by neurotransmitter chemicals, and the amount and type of these chemicals will dictate how strong the input to the neuron is. The function of the biological neural network is replicated by artificial neural networks using weights, biases (a bias is defined as a weight multiplied by a constant input of 1), and activation functions.

The following is a diagrammatic representation of a neural unit:




All a neural network sees are sets of numbers, and it tries to identify a pattern in the data. Through training, the neural network learns to recognize a pattern in the input; however, there are certain specialized architectures that perform better when applied to a certain category of problems than others. A simple neural network architecture consists of three kinds of layer: the **input** layer, the **output** layer, and the **hidden** layer. When there is more than one hidden layer, it is called a **deep neural network**.

The following is a representation of a deep neural network:




In the preceding diagram the circles represent a neuron or in deep learning terms, a node, which is a computation unit. The edges represent the connection between the nodes and hold the connection weight (synapse strength) between the two nodes.

In this chapter, the following recipes will get us started with neural networks:

- Defining the neural network class
- Creating a fully connected network
- Defining the loss function
- Implementing optimizers
- Implementing dropouts
- Implementing functional APIs

# Technical requirements

In this chapter, we will start dealing with image data and learn how a fully connected neural network works. PyTorch has a complementary library called TorchVision, and we will install it before we start with the recipes.

You could use the following `pip` installation command for `torchvision`:

```
pip install torchvision
```

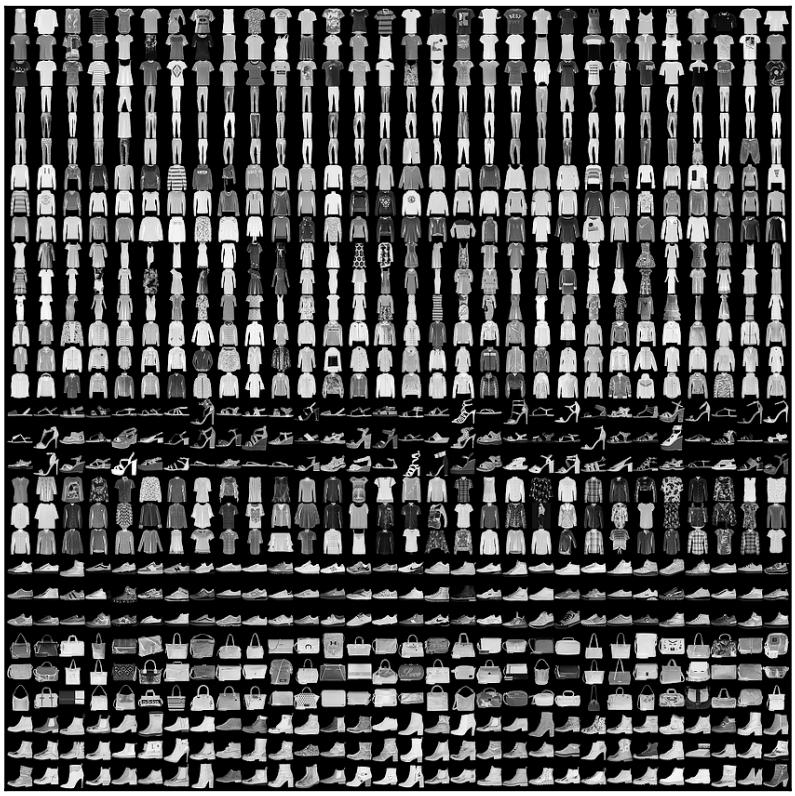
For other installation methods, you can visit `https://pypi.org/project/torchvision/` . The rest of the dependencies from the previous chapter, *Working with Tensors Using PyTorch*, remain the same.

# Defining the neural network class

In this recipe, we will start by understanding some important functions of TorchVision that enable it to deal with image data and process it. We will then define a basic architecture for a neural network by defining a class, and look at the modules and methods available for this. In this recipe, we will be focusing on a fully connected neural network class. Its attributes are the various layers whose purpose is to classify various types of clothes.

We will be using the Fashion–MNIST dataset. This is a dataset of Zalando's article images, consisting of a training set of 60,000 examples and a test set of 10,000 examples. We will take an individual grayscale image 28 x 28 in size and convert it into a vector of 784.

The following is a sample from the dataset:



We will now look at the steps that we need to take to define the network.

# How to do it...

Let's define our network:

1. We will start by setting up the `torch` and `torchvision` imports:

```
>>import torch
>>from torch import nn
>>from torchvision import datasets, transforms
```

2. Next, we will define transforms for the preprocessing of our image data:

```
>>transform = transforms.Compose([transforms.ToTensor(),
                    transforms.Normalize((0.5,), (0.5,)),
                ])
```

3. Let's define the `batch_size` to divide our dataset into chunks to be fed into the model:

```
>>batch_size = 64
```

4. Next, we will pull the dataset from `torchvision` and apply the transform and create batches. For this, we will first create a training dataset:

```
>>trainset = datasets.FashionMNIST('~/.pytorch/F_MNIST_data/',
download=True, train=True, transform=transform)
>>trainloader = torch.utils.data.DataLoader(trainset,
batch_size=batch_size, shuffle=True)
```

5. Now, let's create the `testset`:

```
>>testset = datasets.FashionMNIST('~/.pytorch/F_MNIST_data/',
download=True, train=False, transform=transform)
>>testloader = torch.utils.data.DataLoader(testset,
batch_size=batch_size, shuffle=True)
```

6. Now our main task is to define the neural network class, which has to be a subclass of `nn.Module`:

```
>>class FashionNetwork(nn.Module):
```

7. Next, we define the `init` method for the class:

```
>>def __init__(self):
      super().__init__()
```

8. We need to define the layers for our model within `init`. The first hidden layer looks like the following:

```
>>self.hidden1 = nn.Linear(784, 256)
```

9. Now we will define the second hidden layer:

```
>>self.hidden2 = nn.Linear(256, 128)
```

10. Then we will define our output layer:

```
>>self.output = nn.Linear(128, 10)
```

11. We will define our softmax activation for our last layer:

```
>>self.softmax = nn.Softmax(dim=1)
```

12. Finally, we will define the activation function in the inner layers:

```
>>self.activation = nn.ReLU()
```

With these steps, we have completed our network units.

# How it works...

In this recipe, we started using `torchvision`. There are utilities within `torchvision` to support vision-related tasks. There is a module called `transforms` that helps with a lot of image preprocessing tasks. For the particular case that we are dealing with, an image consisting of 28 x 28 grayscale pixels, we first need to read from the image and convert it into a tensor using a `transforms.ToTensor()` transform. We then make the mean and standard deviation of the pixel values 0.5 and 0.5 respectively so that it becomes easier for the model to train; to do this, we use `transforms.Normalize((0.5,),(0.5,))`. We combine all of the transformations together with `transform.Compose()`.

With the transforms ready, we defined a suitable batch size. A higher batch size means that the model has fewer training steps and learns faster, whereas a high batch size results in high memory requirements.

TorchVision comes with a lot of popular datasets in its `datasets` module; if it's not available on the machine, it will download it for you, pass the transformations, and convert the data into the desired format for the model to train on. In our case, the dataset comes with a training and testing set, and we load them accordingly. We use `torch.utils.data.DataLoader` to load this processed data into batches, along with other operations such as shuffling and loading to the right device—CPU or GPU.

We could define the model class with any name, but what is important is that it is a subclass of `nn.Module` and has `super().__init__()`, which provides the model with a lot of useful methods and attributes and retains knowledge of the architecture.

We use `nn.Linear()` to define fully connected layers by passing in the input and output dimensions. We use a softmax layer for the last layer output because there are 10 output classes. We use ReLU activation in the layers before the output layer to learn nonlinearity in the data. The `hidden1` layer takes 784 inputs units and gives out 256 output units. The `hidden2` phrase outputs 128 units and the output layer has 10 output units representing 10 output classes. The softmax layer converts the activations into probabilities so that it adds to 1 along dimension 1.

# There's more...

There is another method that we can use to define models using `nn.Sequential()` and pass in the required layers without needing to define a class. There are also other transformations that can be applied to the input image that we will explore in the subsequent chapters.

# See also

You can check out more details on transforms at `https://pytorch.org/docs/stable/torchvision/transforms.html`, and you can learn more about defining model classes at `https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py`.

# Creating a fully connected network

In this recipe, we will expand on the class that we defined in the previous recipe, *Defining the neural network class*. In the *Defining the neural network class* recipe, we only created components of the architecture that we needed; now we will look at tying all these pieces together to make a sensible network. The progression for our layers will be from 784 units to 256, then to 128, and finally the output layer of 10 units.

In this recipe, we will work on the network architecture using the components defined in the constructor of our class. We will then complete our network class definition and create its object.

# How to do it...

We will continue with the class definition from the previous section and expand on it:

1. Let's start with the `forward()` method in the class, passing in the input:

```
>>def forward(self, x):
```

2. Now we will move the input to the first hidden layer, with 256 nodes:

```
>>x = self.hidden1(x)
```

3. Next, we pass the outputs from the first hidden layer through the activation function, which in our case is ReLU:

```
>>x = self.activation(x)
```

4. We will repeat the same for the second layer, which has 128 nodes, and pass it through ReLU:

```
>>x = self.hidden2(x)
>>x = self.activation(x)
```

5. Now we pass the last output layer, with 10 output classes:

```
>>x = self.output(x)
```

6. Then we will push the output using the `softmax` function:

```
>>output = self.softmax(x)
```

7. Finally, we return the output tensor:

```
>>return output
```

8. We will then create the network object:

```
>>model = FashionNetwork()
```

9. Let's have a quick look at our model:

```
>>print(model)
>FashionNetwork(
        (hidden1): Linear(in_features=784, out_features=256,
bias=True)
        (hidden2): Linear(in_features=256, out_features=128,
bias=True)
        (output): Linear(in_features=128, out_features=10,
bias=True)
        (softmax): Softmax()
        (activation): ReLU()
 )
```

We have now completed our neural network model for our Fashion–MNIST dataset.

# How it works...

In the recipe, the network is completed by setting up a forward network, wherein we tied together the network components defined in the constructor. A network defined with `nn.Module` needs to have a `forward()` method defined. It takes the input tensor and passes it through the network components defined in the `__init__()` method in the network class, in the sequence of operations defined in the forward method.

The forward method is called automatically when input is passed referring to the name of the model object. The `nn.Module` automatically creates the weight and bias tensors that we'll use in the forward method. The linear unit by itself defines a linear function, such as $xW + B$; to have nonlinear capabilities, we need to insert nonlinear activation functions, and here we use one of the most popular activation functions, ReLU, although you could use other available activation functions in PyTorch.

Our input layer has 784 units (from 28 x 28 pixels), and the first layer has 256 units with ReLU activation, then 128 units with ReLU activation, and finally 10 units with softmax activation. The reason we squish the final layer output through softmax is because we want to have 1 output class with a higher probability than all the other classes, and the sum of the output probabilities should equal 1. The softmax function has a parameter `dim=1` that ensures that softmax is taken across the columns of the output. We then create an object using the model class and print the details of the class using `print(model)`.

## There's more...

We can define the network architecture without defining a network class using the `nn.Sequential` module, and it is important to ensure that the sequence of operation in the `forward` method is ordered properly, although the sequence doesn't matter in `__init__`. You can use `nn.Tanh` for tanh activation. You can access the weight and bias tensors from the model object with `model.hidden.weight` and `model.hidden.bias`.

## See also

You can check the official documentation for `nn.Module` and `nn.Sequential` at `https://pytorch.org/docs/stable/nn.html`.

# Defining the loss function

A machine learning model, when being trained, may have some deviation between the predicted output and the actual output, and this difference is called the **error** of the model. The function that lets us calculate this error is called the **loss function**, or **error function**. This function provides a metric to evaluate all possible solutions and choose the most optimized model. The loss function has to be able to reduce all attributes of the model down to a single number so that an improvement in that loss function value is representative of a better model.

In this recipe, we will define a loss function for our fashion dataset using the loss function available in PyTorch.

# How to do it...

Let's define our loss function:

1. First, we will modify our existing network architecture to the output log of `softmax` instead of `softmax`, starting with the `__init__` method in the network constructor:

```
>>self.log_softmax = nn.LogSoftmax()
```

2. Next, we will make the same change in the `forward` method of the neural network:

```
>>output = self.log_softmax(x)
```

3. So now our new class looks as follows:

```
>>class FashionNetwork(nn.Module):
      def __init__(self):
          super().__init__()
          self.hidden1 = nn.Linear(784, 256)
          self.hidden2 = nn.Linear(256, 128)
          self.output = nn.Linear(128, 10)
          self.log_softmax = nn.LogSoftmax()
          self.activation = nn.ReLU()
      def forward(self, x):
          x = self.hidden1(x)
          x = self.activation(x)
          x = self.hidden2(x)
          x = self.activation(x)
          x = self.output(x)
          output = self.log_softmax(x)
          return output
```

4. We define the model object as follows:

```
>>model = FashionNetwork()
>>model
>>FashionNetwork(
     (hidden1): Linear(in_features=784, out_features=256,
bias=True)
     (hidden2): Linear(in_features=256, out_features=128,
bias=True)
     (output): Linear(in_features=128, out_features=10, bias=True)
     (log_softmax): LogSoftmax()
     (activation): ReLU()
 )
```

5. Now, we will define our loss function; we will use negative log likelihood loss for this:

```
>criterion = nn.NLLLoss()
```

We now have our loss function ready.

# How it works...

In this recipe, we replaced softmax with log softmax so that we could then use the log of probabilities over probabilities, which has nice theoretic interpretations. There are various reasons for doing this, including improved numerical performance and gradient optimization. These advantages can be extremely important when training a model that can be computationally challenging and expensive. Furthermore, it has a high penalizing effect when it is not predicting the correct class.

We therefore use negative log likelihood when dealing with log softmax, as softmax is not compatible. It is useful in classification between *n* number of classes. The log would ensure that we are not dealing with very small values between 0 and 1, and negative values would ensure that a logarithm of probability that is less than 1 is nonzero. Our goal would be to reduce this negative log loss error function. In PyTorch, the loss function is called a **criterion**, and so we named our loss function `criterion`.

# There's more...

We can provide an optional argument, `weight`, that has to be a 1D tensor that assigns weights to each of the output classes to deal with unbalanced training sets.

# See also

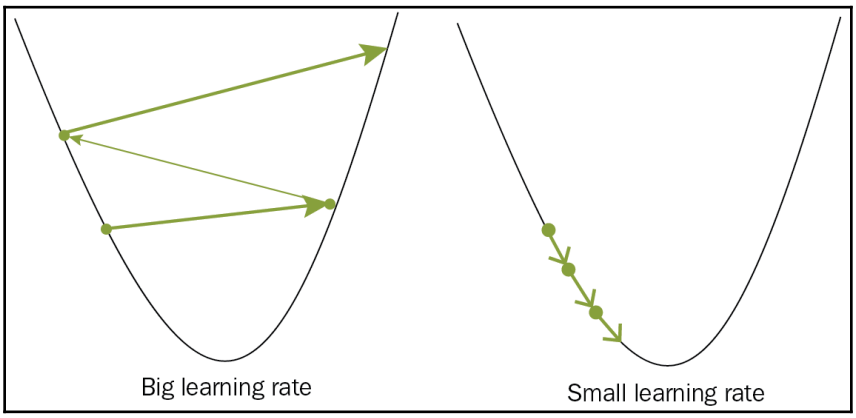
You can look at the official documentation for more loss functions at `https://pytorch.org/docs/master/nn.html#loss-functions`.

# Implementing optimizers

In this recipe, we will be learning about optimizers. In the previous recipe, *Defining the loss function*, we spoke of errors and error functions, and learned that, for us to get a good model, we need to minimize the errors that are calculated. **Backpropagation** is a method by which the neural networks learn from errors; the errors are used to modify weights in such a way that the errors are minimized. Optimization functions are responsible for modifying weights to reduce the error. Optimization functions calculate the partial derivative of errors with respect to weights. The derivative shows the direction of a positive slope, and so we need to reverse the direction of the gradient. The **optimizer** function combines the model parameters and loss function to iteratively modify the model parameters to reduce the model error. Optimizers can be thought of as fiddling with the model weights to get the best possible model based on the difference in prediction from the model and the actual output, and the loss function acts as a guide by indicating when the optimizer is going right or wrong.

The learning rate is a hyperparameter of the optimizer, which controls the amount by which the weights are updated. The learning rate ensures that the weights are not updated by a huge amount so that the algorithm fails to converge at all and the error gets bigger and bigger; however at the same time, the updating of the weight should not be so low that it takes forever to reach the minimum of the cost function/error function.

The following shows the effects of the learning rate:




In this recipe, we will learn about using optimizer functions in PyTorch, as well as some common optimization functions and how to handle learning rates.

# How to do it...

In this section, we start from the juncture where we last left our code in the previous section, at the point where we defined the criterion:

1. We will start by importing the `optim` module:

```
>>from torch import optim
```

2. Next, we will create an `optimizer` object. We will use the `Adam` optimizer and pass model parameters:

```
>>optimizer = optim.Adam(model.parameters())
```

3. To check for the defaults of the optimizer, you can do the following:

```
>>optimizer.defaults
>>{'lr': 0.001,
 'betas': (0.9, 0.999),
 'eps': 1e-08,
 'weight_decay': 0,
 'amsgrad': False}
```

4. You can also add the learning rate as an additional parameter:

```
>>optimizer = optim.Adam(model.parameters(), lr=3e-3)
```

5. Now we will start training our model, starting with the number of epochs:

```
>>epoch = 10
```

6. We will then start the loop:

```
>>for _ in range(epoch):
```

7. We initialize `running_loss` as `0`:

```
>>running_loss = 0
```

8. We will iterate through each image in training the image loader, which we defined in an earlier recipe in this chapter: *Defining the neural network class*:

```
>>for image, label in trainloader:
```

9. We then reset the gradients to zero:

```
>>optimizer.zero_grad()
```

10. Next, we will reshape the image:

```
>>image = image.view(image.shape[0],-1)
```

11. Then we get the prediction from the model:

```
>>pred = model(image)
```

12. Then we calculate the loss/error:

```
>>loss = criterion(pred, label)
```

13. Then we call the `.backward()` method on the loss:

```
>>loss.backward()
```

14. Then we call the `.step()` method on the optimizer:

```
>>optimizer.step()
```

15. Then we append to the running loss:

```
>>running_loss += loss.item()
```

16. Finally, we will print the loss after each epoch:

```
>>else:
    >>print(f'Training loss: {running_loss/len(trainloader):.4f}')
```

The following is a sample output:

```
Training loss: 0.4978
Training loss: 0.3851
Training loss: 0.3498
Training loss: 0.3278
Training loss: 0.3098
Training loss: 0.2980
Training loss: 0.2871
Training loss: 0.2798
Training loss: 0.2717
Training loss: 0.2596
```

Now we have completed the training.

# How it works...

In this recipe, we started by defining the optimizer using an `Adam` optimizer, and then we set a learning rate for the optimizer and had a look at the default parameters. We set an epoch of `10` and started iterations for each epoch, setting `running_loss` to 0 on each iteration and iterating over each image within the epoch (the number of times the model sees the dataset). We started by clearing the gradients using the `.zero_grad()` method. PyTorch accumulates gradients on each backward pass, which is useful in some cases, and so it was imported to zero out the gradient to properly update the model parameters.

Next, we reshaped the image by flattening each batch of 64 images (consisting of 28 x 28 pixels in each image) to 784, thereby changing the tensor shape from 64 x 28 x 28 to 64 x 784, as our model expects this shape for the input. Next, we sent this input over to the model and got the output predictions for the batch from the model, and then passed it to the loss function, also called `criterion`; there, it assessed the difference between the predicted and the actual class.

The `loss.backward()` function calculated the gradient—that is, the partial derivative of the error with respect to the weights—and we called the `optimizer.step()` function to update the weights of the model to adapt to the error that was evaluated. The `.item()` method pulled a scalar out of a single element tensor, and so with `loss.item()` we get a scalar value of `error` from the batch, accumulate it to the losses through all the batches, and finally print the loss at the end of the epoch.

# There's more...

We can use a callback function called `closure` as a parameter for `.step(closure)` to calculate the loss and update the weights by passing in a function as a parameter. You could also explore other optimizer functions, such as Adadelta, Adagrad, SGD, and so on, which are available with PyTorch.

# See also

You can read more about optimizers at `https://pytorch.org/docs/stable/optim.html#torch.optim.Optimizer`.

# Implementing dropouts

In this recipe, we will look at implementing dropouts. One of the more common phenomena that we might encounter while training a neural network model, or any machine learning model in general, is overfitting. Overfitting happens when a model learns the data that is given to it for training rather than generalizing on the solution space—that is, it learns the minute details and noises of the training data, instead of grasping the bigger picture, and so performs poorly on new data. Regularization is the process of preventing models from overfitting.

Using a dropout is one of the most popular regularization techniques in neural networks, in which randomly selected neurons are turned off while training—that is, the contribution of neurons is temporarily removed from the forward pass and the backward pass doesn't affect the weights, so that no single neuron or subset of neurons gets all the decisive power of the model; rather, all the neurons are forced to make active contributions to predictions.

Dropouts can be intuitively understood as creating a large number of ensemble models, learning to capture various features under one big definition of a model.

In this recipe, we will look at how to add dropouts to our model definition to improve the overall model performance by preventing overfitting. It should be remembered that dropouts are to be applied only while training; however, when testing and during the actual prediction, we want all of the neurons to make contributions.

# How to do it...

In this section, we will learn how to add dropouts to our initial model class, called `FashionNetwork`:

1. We will start with our initial model definition:

```
>>class FashionNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden1 = nn.Linear(784, 256)
        self.hidden2 = nn.Linear(256, 128)
        self.output = nn.Linear(128, 10)
        self.log_softmax = nn.LogSoftmax()
        self.activation = nn.ReLU()
    def forward(self, x):
        x = self.hidden1(x)
        x = self.activation(x)
        x = self.hidden2(x)
```

```
        x = self.activation(x)
        x = self.output(x)
        output = self.log_softmax(x)
        return output
```

2. Then we will add a dropout to our model `__init__`:

```
>>self.drop = nn.Dropout(p=0.25)
```

Our updated `__init__()` looks as follows:

```
>>def __init__(self):
        super().__init__()
        self.hidden1 = nn.Linear(784, 256)
        self.hidden2 = nn.Linear(256, 128)
        self.output = nn.Linear(128, 10)
        self.log_softmax = nn.LogSoftmax()
        self.activation = nn.ReLU()
        self.drop = nn.Dropout(p=0.25)
```

3. Now, we will add dropouts in our `forward()` method:

```
>>def forward(self, x):
        x = self.hidden1(x)
        x = self.activation(x)
        x = self.drop(x)
        x = self.hidden2(x)
        x = self.activation(x)
        x = self.drop(x)
        x = self.output(x)
        output = self.log_softmax(x)
        return output
```

We now have a network with dropouts.

# How it works...

In this recipe, we altered the `__init__()` method to add the dropout layer with a dropout probability of 0.25, which means that 25% of the neurons in the layer where this dropout is applied will be turned off randomly. Then, we edited our `forward` function, applied it to the first hidden layer with 256 units in it, and then we applied the dropout on the second layer, which has 128 units. We applied the activation in both the layers after going through the activation functions. We have to keep in mind that dropouts must be applied only on hidden layers in order to prevent us from losing the input data and missing outputs.

## There's more...

We can disable dropouts by calling `model.eval()` and enable dropouts using `model.train()`.

## See also

You can learn more about dropouts at `https://arxiv.org/abs/1207.0580`.

# Implementing functional APIs

In this recipe, we will explore functional APIs in PyTorch; doing so will allow us to write cleaner and more concise network architectures and components. We will be looking at functional APIs and defining models, or a part of a model, with functional APIs.

## How to do it...

In the following steps, we use our existing neural network class definition and then rewrite it using functional APIs:

1. We will start by making an import:

```
>>import torch.nn.functional as F
```

2. Then we define our `FashionNetwork` class with `F.relu()` and `F.log_softmax()`:

```
>>class FashionNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden1 = nn.Linear(784,256)
        self.hidden2 = nn.Linear(256,128)
        self.output = nn.Linear(128,10)
    def forward(self,x):
        x = F.relu(self.hidden1(x))
        x = F.relu(self.hidden2(x))
        x = F.log_softmax(self.output(x))
        return x
```

We redefined our model with functional APIs

# How it works...

In this recipe, we defined the exact same network as before, but replaced the activation function and the log softmax with `function.relu` and `function.log_softmax`, which makes our code look a lot cleaner and more concise.

# There' s more...

You could use functional APIs for linear layers by using `functional.linear()` and `functional.dropout()` to control dropouts, but you must take care to pass the model state to indicate whether it is in training or evaluation/prediction mode.

# See also

You can learn more about functional APIs at `https://pytorch.org/docs/stable/nn.html#torch-nn-functional`.

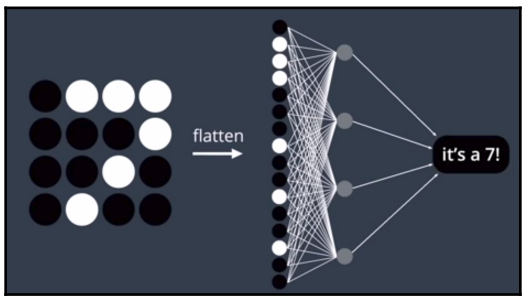
# 3
# Convolutional Neural Networks for Computer Vision

In this chapter, we will be learning about **convolutional neural networks** (**CNNs**). This is a different class of neural network to the ones discussed in the previous chapters. CNNs have been hugely successful in the domain of computer vision, and as we learn more about them, we will be able to appreciate the reasons for this.

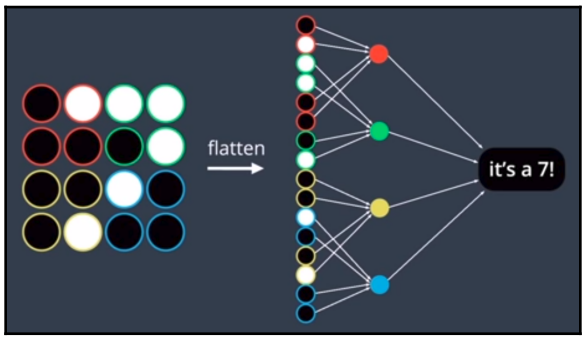
CNNs are a specialized kind of network that can take in images as tensors. A colored image consists of three color channels, red, green, and blue, referred to as RGB. These 2D-matrix channels are stacked to form colored images as we know them; the variations in the values of each channel give rise to different colors. A CNN takes in images as three separate stacked strata of color, one on top of the other.

An image gets its meaning from a set pixel in the neighborhood, but a single pixel doesn't hold much information about the entire image. In a fully connected neural network, which is also called a dense layer, every node from one layer is connected to every other node in the subsequent layer. A CNN leverages the spatial structure between the pixels to reduce the number of connections between two layers, significantly improving the speed of training while at the same time reducing the model parameters.

Here is an image showing a fully connected network:



Compare the previous image with the following one, which shows a convolutional network:




A CNN picks up features from an input image using a filter; a CNN with a sufficient number of filters detects various features in the image. These filters become more and more sophisticated in detecting complex features as we move more and more toward the later layers. Convolutional networks use these filters and map them one by one to create a map of feature occurrences.

In this chapter, we will be covering the following recipes:

- Exploring convolutions
- Exploring pooling
- Exploring transform
- Performing data augmentation
- Loading image data
- Defining CNN architecture
- Training an image classifier

# Technical requirements

In this chapter, you will need TorchVision, which we installed in the previous chapter. It is preferable for you to run the codes in these recipes on a GPU-enabled machine.

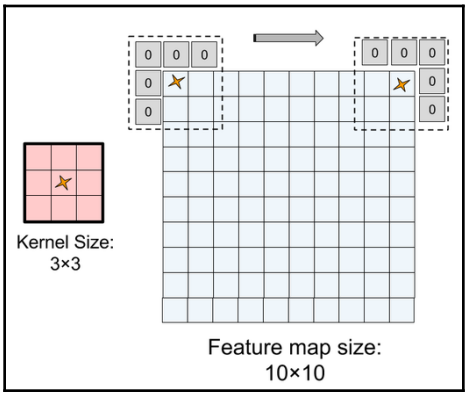
# Exploring convolutions

Convolutions are a component within CNNs. They are defined as a layer within the CNNs. In a convolution layer, we slide a filter matrix over the entire image matrix from left to right and from top to bottom, and we take the dot product of the filter, with this patch spanning the size of the filter over the image channel. If the two matrices have high values in the same positions, the dot product's output will be high, and vice versa. The output of the dot product is a scalar value that identifies the correlation between the pixel pattern in the image and the pixel pattern expressed by the filter. Different filters detect different features from the image and at various levels of complexity.

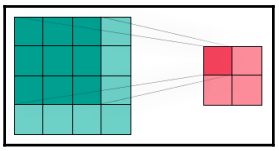
We need to understand two more key elements of CNNs, which are as follows:

- **Stride**: This is the number of pixels that we shift both horizontally and vertically before applying convolution networks using a filter on the next patch of the image.
- **Padding**: This is the strategy that we apply to the edges of an image while we convolve, depending on whether we want to keep the dimensions of the tensors the same after convolution or only apply convolution where the filter fits properly with the input image. If we want to keep the dimensions the same, then we need to zero pad the edge so that the original dimensions match with the output after convolution. This is called **same padding**. But if we don't want to preserve the original dimensions, then the places where the filter doesn't fit completely are truncated, which is called **valid padding**.

Here is the diagrammatic representation of these two paddings:



The following image shows an example of valid padding:



In this recipe, we will learn how to use convolution neural networks in PyTorch.

# How to do it...

In this recipe, we will explore convolutions:

1. First, we will import the torch `modules` that we need:

```
>>import torch
>>import torch.nn as nn
```

2. Next, we will apply 2D convolution to an image:

```
>>nn.Conv2d(3, 16, 3)
```

   This creates the following convolution layer:

```
Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1))
```

3. We then add padding of the desired size to the edge of an image:

```
>>nn.Conv2d(3, 16, 3, padding=1)
```

   This creates the following convolution layer:

```
Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
```

4. We can then create a non square kernel (filter) by using the following code:

```
>>nn.Conv2d(3, 16, (3,4), padding=1)
```

   This creates the following convolution layer:

```
Conv2d(3, 16, kernel_size=(3, 4), stride=(1, 1),
padding=(1, 1))
```

5. We can then add stride to our convolution using the following code:

```
>>nn.Conv2d(3, 16, 3, stride=2)
```

   This creates the following convolution layer:

```
Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2))
```

6. We can have unequal stride and padding along the horizontal and vertical directions:

```
>>nn.Conv2d(3, 16, (3,4), stride=(3,3), padding=(1,2))
```

   This creates the following convolution layer:

```
Conv2d(3, 16, kernel_size=(3, 4), stride=(3, 3),
padding=(1, 2))
```

With this recipe, we have learned how to use convolution in PyTorch.

# How it works...

In this recipe, we looked at multiple ways of creating a 2D convolution, wherein the first parameter is the number of channels in a given input image, which will be `3` for a color image and `1` for a grayscale image. The second parameter is the number of output channels—in other words, the number of filters that we want from the given layer. The third parameter is the kernel size—which is the size of the kernel—or the patch size of the image to be convoluted with a filter.

We then created a `Con2d` object and passed the input to the 2D convolutional layer to get the output. With `nn.Conv2d(3, 16, 3)`, we created a convolutional layer, which takes in an input of 3 channels and outputs 16 channels. This layer has a square kernel of size 3 x 3 with a default stride of 1 in its height and width. We can add padding using the `padding` parameter, which can have an integer or a tuple value. Here, the integer value will create equal paddings for height and width, and a tuple value will have different paddings for height and width—this is true for the kernel size as well as the stride.

## There's more...

You can have valid padding by setting the `padding` argument to `0`, which is set by default. You can also change the zero padding into circular padding by changing the `padding_mode` argument. You can add or remove bias using the `bias` Boolean argument, which by default is `True`.

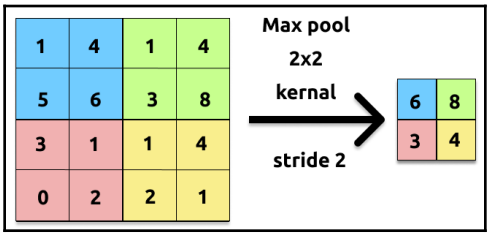
## See also

You can learn about other arguments for PyTorch convolutions at `https://PyTorch.org/docs/stable/nn.html#torch.nn.Conv2d`.

# Exploring pooling

Now we move on to the next crucial layer of CNNs—the pooling layer. So far, we have been dealing with images without changing the spatial dimensions of the frames (considering the same padding); instead, we have been increasing the number of channels/filters. The pooling layer is used to reduce the spatial dimension of an input, preserving its depth. As we move from the initial layer to the later layers in a CNN, we want to identify more conceptual meaning in the image compared to actual pixel by pixel information, and so we want to identify and keep key pieces of information from the input and throw away the rest. A pooling layer helps us do that.

Here is a diagrammatic illustration of max pooling:




Here are the main reasons to use a pooling layer:

- **Reduction in the number of computations**: We get better computational performance by reducing the spatial dimensions of the input without losing out on the filters, and so we reduce the time needed to train, as well as the computational resources.

- **Prevent overfitting**: With reduced spatial dimensions, we reduce the number of parameters the model has, which in turn reduces the model complexity and helps us generalize better.
- **Positional invariance**: This allows the CNN to capture the features within an image, irrespective of where the feature is located in a given image. Say that we are trying to build a classifier to detect mangoes. It doesn't matter whether the mango is in the center, top-left, bottom-right, or wherever in the image—it needs to be detected. The pooling layer helps us with this.

There are many types of pooling, such as max pooling, average pooling, sum pooling, and so on; however, max pooling is the most popular. In the same way that we dealt with a convolutional layer, we will define a window and apply the desired pooling operation in that window. We will slide the window horizontally and vertically, as defined by the stride of the layer.

# How to do it...

In this recipe, we will look at how to implement a pooling layer in PyTorch:

1. First, let's make the imports:

```
>>import torch
>>import torch.nn as nn
```

2. Then, we use the pooling classes defined in the `nn` modules, as follows:

```
>>max_pool = nn.MaxPool2d(3, stride=1)
```

3. Now, let's define a tensor to perform the pooling on:

```
>>a = torch.FloatTensor(3,5,5).random_(0, 10)
>>a
```

This gives us the following output:

```
tensor([[[2., 8., 6., 8., 3.],
         [6., 6., 7., 6., 6.],
         [2., 0., 8., 8., 8.],
         [2., 0., 3., 5., 7.],
         [9., 7., 8., 2., 1.]],

        [[1., 8., 6., 7., 3.],
         [0., 1., 2., 9., 4.],
         [1., 2., 5., 0., 1.],
```

```
          [8., 2., 8., 3., 1.],
          [5., 4., 0., 5., 2.]],

         [[1., 6., 2., 6., 1.],
          [4., 0., 0., 6., 6.],
          [4., 2., 2., 3., 2.],
          [1., 0., 1., 7., 1.],
          [8., 1., 0., 5., 4.]]])
```

4. Now, we apply pooling to the tensor:

```
>>max_pool(a)
```

This gives us the following output:

```
tensor([[[8., 8., 8.],
     [8., 8., 8.],
     [9., 8., 8.]],

    [[8., 9., 9.],
     [8., 9., 9.],
     [8., 8., 8.]],

    [[6., 6., 6.],
     [4., 7., 7.],
     [8., 7., 7.]]])
```

5. We can now try average pooling in a similar fashion:

```
>>avg_pool = nn.AvgPool2d(3, stride=1)
```

6. We then apply average pooling, as before:

```
>>avg_pool(a)
```

This gives us the following output:

```
tensor([[[5.0000, 6.3333, 6.6667],
     [3.7778, 4.7778, 6.4444],
     [4.3333, 4.5556, 5.5556]],

    [[2.8889, 4.4444, 4.1111],
     [3.2222, 3.5556, 3.6667],
     [3.8889, 3.2222, 2.7778]],

    [[2.3333, 3.0000, 3.1111],
     [1.5556, 2.3333, 3.1111],
     [2.1111, 2.3333, 2.7778]]])
```

With this recipe, we have learned about the pooling operation in PyTorch.

## How it works...

In the preceding code, we worked through an example of a tensor to see a pooling layer in action. We used a square kernel of size 3 x 3. The first application of pooling happened on the patch [0,0,0] to [0,3,3]. Since the stride is 1, the next patch to be operated on was [0,0,1] to [0,3,4]. Once it met the horizontal end, the tensor right below was operated on.
Both `nn.MaxPool2d(3, stride=1)` and `nn.AvgPool2d(3, stride=1)` created a max and average pool square kernel of size `3x3` with a stride of `1`, which was applied on a random tensor, `a`.

## There's more...

In this recipe, we looked at a square kernel, but we can choose to use a non square kernel and strides, just like we did for convolutions. There is another popular pooling method known as global average pooling, which can be achieved by average pooling, by passing in the dimensions of the input; for example, `avg_pool2d(a, a.size()[2:]0)`.

## See also

You can find out more about pooling and the various types of pooling at `https://PyTorch.org/docs/stable/nn.html#pooling-layers`.

# Exploring transforms

PyTorch cannot process an image pixel directly and needs to have the contents as tensors. To get around this, `torchvision`, being a specialized library for vision and image-related tasks, provides a module called `transform`, which provides APIs for converting pixels into tensors, normalizing standard scaling, and so on. In this recipe, we will explore various methods in the transform module. Because of this, you need to have `torchvision` installed to go through this recipe.

# How to do it...

In this section, we will explore the various transforms in `torchvision`:

1. We will start by importing `torchvision`:

```
>>from torchvision import transforms
```

2. Let's create a tensor from the image:

```
>>transforms.ToTensor()
```

3. Next, let's normalize the image tensor:

```
>>transforms.Normalize((0.5,),(0.5,))
```

4. To resize an image, we will use the following method:

```
>>transforms.Resize(10)
```

   We can also use the following:

```
>>transforms.Resize((10,10))
```

5. Then, we use a transform to crop the image:

```
>>transforms.CenterCrop(10)
```

   We can also use the following:

```
>>transforms.CenterCrop((10, 10))
```

6. We could use transforms to pad the image tensors:

```
>>transforms.Pad(1, 0)
```

   We can also use the following:

```
>>transforms.Pad((1, 2), 1)
```

   We can also do the following if we prefer:

```
>>transforms.Pad((1, 2, 2, 3), padding_mode='reflect')
```

7. Then, we chain multiple transforms:

```
>>transforms.Compose([
     transforms.CenterCrop(10),
     transforms.ToTensor(),
  ])
```

In this recipe, we learned about some of the transforms that are used in `torchvision`.

# How it works...

In the preceding code snippets, we looked at the various transforms that are available in `torchvision`. These allow us to take input images and format them into tensors of the desired dimensions and properties, which can then be fed into torch models. The first method that we looked at was the `toTensor()` method, which transforms a given input image into a tensor. We could then normalize this input image tensor using the `Normalize()` method. The `Normalize()` method takes in two tuples, where the first tuple is the sequence of the means of each channel in the input image and the second tuple is the sequence of the standard deviation for each channel.

Furthermore, we could resize a given image into the desired dimensions using the `Resize()` method, which, if given an integer, would match it with the length of the smaller edge, and if given a tuple, would match the height and width of the image. There would be certain cases where the crucial information regarding the image is in its center, and in such cases, it would be okay to crop and consider only the center of the given image; for this, you could use the `CenterCrop()` method. Then, we passed in an integer to crop a square from the center or to pass a sequence matching the height and width to `CenterCrop()`.

Another important task is to pad the image to match certain dimensions. For this, we use the `Pad()` method. We pass in the padding size as the integer for equal-sized padding on all sides or a sequence consisting of two elements for the padding size corresponding to the left/right and top/bottom, respectively. Furthermore, we could pass in the padding size for the left, top, right, and bottom sides as a sequence consisting of four elements. We then provided a fill value as an integer, and if it's a tuple of three elements, it's used as pad values for the R, G, and B channels, respectively. Apart from these, the `Pad()` method also has a `padding_mode` parameter, which takes in the following possibilities:

- `constant`: Pads with a fill value provided
- `edge`: Pads with the value at the edge of the image

- `reflect`: Pads with a reflection of the image, excluding the edge pixel
- `symmetric`: Pads with a reflection of the image, including the edge pixel

In the end, we looked at the `Compose()` transform, which combined the various transforms to build a transformation pipeline by passing in a list of transform objects as a parameter.

## There's more...

There are functional APIs for transformations in the `transforms.functional` module. They help us build complex transformation pipelines by providing fine-grained control over transformations.

There are other useful transformations, such as grayscale transformations, which use `Grayscale()` with the number of output channels as a parameter. We will explore more transforms in the next section.

## See also

You can read more about functional transformations at `https://PyTorch.org/docs/stable/torchvision/transforms.html#functional-transforms`.

# Performing data augmentation

In this recipe, we will learn about data augmentation with torch. Data augmentation is an important technique in deep learning and computer vision. For any model dealing with deep learning or computer vision, the amount of data available is crucial to see how well the model performs. Data augmentation prevents models from memorizing the limited amount of data rather than making generalizations about the observed data. Data augmentation increases the diversity of data for training the model by creating variations from the original images without actually collecting new data.

Oftentimes, the amount of light, brightness, orientation, or color variations doesn't make a difference to the inferences that are made by a model; however, when the model is deployed in the real world, the input data may have these variations. It is useful for the model to know that the decision it makes has to be invariant with respect to these variations in the input, and so data augmentation improves model performance. In this recipe, we will use PyTorch's `transform` module to preform data augmentation.

# How to do it...

In order to make the best of this recipe, you should complete the *Exploring transforms* recipe, as this recipe is a continuation of our work with transforms. In this recipe, we will look at some of the popular data augmentations we can perform using the `transform` module in `torchvision`:

1. We will start by importing `torchvision`:

```
>>import torchvision
```

2. We will then crop a section of the image at random:

```
>>transforms.RandomCrop(10)
```

   We could also use the following:

```
>>transforms.RandomCrop((10,20))
```

3. We could flip the image horizontally by using the following:

```
>>transforms.RandomHorizontalFlip(p=0.3)
```

4. We could also flip it vertically:

```
>>transforms.RandomVerticalFlip(p=0.3)
```

5. Try adding brightness, contrast, saturation, and hue variations:

```
>>transforms.ColorJitter(0.25, 0.25, 0.25, 0.25)
```

6. Next, let's add rotational variation:

```
>>transforms.RandomRotation(10)
```

7. Finally, we will compose all the transformations:

```
>>transforms.Compose([
     transforms.RandomRotation(10),
     transforms.ToTensor(),
])
```

In this recipe, we created transforms on the data to create more data from existing data.

# How it works...

In this recipe, we saw how we could add variations to our data by performing certain transformations that are meaningful for the problem at hand. We have to be careful when picking the right data augmentation that mimics the kind of image variations that we would get in real life. For instance, when building a car classifier, it would make sense to augment data with variations in colors and brightness, or flipping the car image horizontally, and so on; however, it would not make sense for us to augment data with car images that are flipped vertically, unless we are dealing with a problem where a car is turned upside down.

In this recipe, we tried cropping the image in a random place so that if the entire image of an object isn't available but a portion is, then our model would be able to detect the object. We should include the cropped image size as an integer or a tuple of a particular height and width. Then, we flipped our image horizontally and passed in a probability for the random horizontal flip and vertical flip. We then created variations in the color, contrast, saturation, and hue of the image using the `ColorJitter()` method.

We controlled the amount of variation in each of them by setting the parameter, where the color, contrast, and saturation vary between the [max(0, 1 - parameter), 1 + parameter] values and the hue varies between [-hue, hue], where the hue is between 0 and 0.5. We also added a random rotation to the images and provided the maximum angle of rotation. Finally, after we picked the right data augmentation strategy, we added it to `transforms.compose()`.

# There's more...

We can also custom define the transform that we need for our image data. For this, we would use `transforms.Lambda()` and pass in a function or a lambda for the custom transformation we want.

# See also

You can learn about other transformations, such as affine transformation and more, at `https://PyTorch.org/docs/stable/torchvision/transforms.html`.

# Loading image data

In this recipe, we will look at how to load image data from files into tensors. In this recipe, we will use the CIFAR-10 dataset, which consists of 60,000 32 x 32 pixel colored images for each of the 10 classes in the dataset. These classes are Airplane, Automobile, Bird, Cat, Deer, Dog, Frog, Horse, Ship, and Truck.

# Getting ready

We will use `torchvision` to load our data. CIFAR-10 is available as a dataset within `torchvision`. You should have installed `torchvision` by this stage; if not, you can use the following code to install it:

```
pip install torchvision==0.x.x
```

with this set up, we are good to go for this recipe.

# How to do it...

In this recipe, we will load the CIFAR-10 dataset in PyTorch:

1. We will import the `datasets` module from `torchvision`:

```
>>from torchvision import datasets
```

2. Then, we will import the `transforms` module:

```
>>from torchvision import transforms
```

3. We will then create a transformation pipeline:

```
>>transformations = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(20),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5),(0.5, 0.5, 0.5))
])
```

4. Next, we will use the datasets module to create the training dataset:

```
>>train_data = datasets.CIFAR10('CIFAR10', train=True,
download=True, transform=transformations)
```

5. Similarly, we will create a test dataset:

```
>>test_data = datasets.CIFAR10('CIFAR10', train=False,
download=True, transform=transformations)
```

6. Now, we can check the lengths of the training and test datasets:

```
>>len(train_data), len(test_data)
(50000, 10000)
```

7. We will now create a validation set from our training set; for this, we will make an import from the torch module:

```
>>from torch.utils.data.sampler import SubsetRandomSampler
```

8. We will select 20% of the training data as validation data:

```
>>validation_size = 0.2
```

9. Now, we will import `numpy`:

```
>>import numpy as np
```

10. We will then get the size of the training data:

```
>>training_size = len(train_data)
```

11. Next, we create a list of indices:

```
>>indices = list(range(training_size))
```

12. We will then shuffle the list of indices:

```
>>np.random.shuffle(indices)
```

13. After this, we will get the index to split the validation and training datasets:

```
>>index_split = int(np.floor(training_size * validation_size))
```

14. Then, we will get the training and validation set indices:

```
>>validation_indices, training_indices = indices[:index_split],
indices[index_split:]
```

15. We will now use the subset random sampler from torch:

```
>>training_sample = SubsetRandomSampler(training_indices)
>>validation_sample = SubsetRandomSampler(validation_indices)
```

16. Next, we will split the datasets into batches. We will set the batch size to 16:

```
>>batch_size = 16
```

17. We will then use the `dataloader` module in PyTorch to load our data into batches:

```
>>from torch.utils.data.dataloader import DataLoader
```

18. Then, we will create training, validation, and test dataset batches:

```
>>train_loader = DataLoader(train_data, batch_size=batch_size,
sampler=training_sample)
>>valid_loader = DataLoader(train_data, batch_size=batch_size,
sampler=validation_sample)
>>test_loader = DataLoader(train_data, batch_size=batch_size)
```

With this, we have loaded our data and preprocessed it so that it is ready to be sent to the model for training.

# How it works...

In this recipe, we used the `datasets` module in PyTorch to get the CIFAR10 dataset. We then defined the transformations that would make sense for the images in the dataset, which are images of animals corresponding to 10 different classes. We performed a horizontal flip for some of the images at random and also added rotation to some of the images at random, with a range of -20 to 20 degrees.

However, we didn't add a vertical flip, since we don't anticipate having an upside-down image of animals to feed into the model in the evaluation phase. After that, we converted the images into tensors using the `ToTensor()` transform. Once the tensors were prepared, we used the `Normalize()` transform to set the mean and standard deviation for each of the red, green, and blue channels, respectively. Following this, we used the `CIFAR10()` method in the datasets to use the CIFAR10 dataset. Then, we set the `download` parameter to `True` so that if the dataset is not present in the root directory, `CIFAR10` (the first argument), then it will be downloaded and kept in that directory.

For the training data, we set the `train` parameter to `True` and passed the transformations that were to be applied to the data using the `transform` parameter. This allowed us to create images on the fly without explicitly creating new images. Now, to prepare the test data, we set the `train` argument to `False`. We set the size of the training and test dataset to be 50,000 and 10,000, respectively. Then, we prepared the validation set from the training set using 20% of the training set, as defined by `validation_size`. We randomly picked 20% of the training set to create a validation set so that the validation set is not skewed to a certain class of animal. We then took the size of the training set and prepared a list of indices using `range()` in Python.

We then shuffled a list of indices using the `random.shuffle()` method in `numpy`. Once the list of indices was randomized, we moved the first 20% of the indices to the validation set and the remaining 80% of the indices to the training set. We found the split index by multiplying the original training size with the percentage of the original training set to be used as a validation set. We used `split_index` for the split. We then used the `SubsetRandomSampler()` method in `torch.utils.data` to sample the elements randomly from a given list of indices, without replacement. Finally, we used `DataLoader()` to combine a dataset and sampler to provide an iterable over the dataset. We then used the dataloader for the training, validation, and test sets to iterate over the data while training the model.

# There's more...

There are many more functionalities in the `DataLoader()` module—for instance, `DataLoader()` can be used for multiprocess data loading, and `num_workers` controls the number of subprocess that are to be used while loading the data. In our example, we have used the default value of `0`, which means that the data is loaded in the main process, which is ideal for small datasets, and gives us more readable error traces.

# See also

You can read more about data loading utils at `https://PyTorch.org/docs/stable/data.html#module-torch.utils.data`.

# Defining the CNN architecture

So far in this chapter, we have been looking at the different components of a CNN and how to load data from the dataset into a format that can be fed into a CNN model. In this recipe, we will define the CNN model architecture from the components that we have seen so far to complete the model. This is very similar to the fully connected neural network we looked at in `Chapter 2`, *Dealing with Neural Networks*. To better understand this recipe, it would be a good idea to revise the model definition of a fully connected neural network from `Chapter 2`, *Dealing with Neural Networks*. We will build our model for image classification on the CIFAR10 dataset, which we discussed in the *Loading image data* recipe.

# How to do it...

We will complete the model class definition in this recipe:

1. First, we will import the `nn.Module` and `torch` functional APIs:

```
>>import torch.nn as nn
>>import torch.nn.functional as F
```

2. We will then write a class that inherits from `nn.Module`:

```
>>class CNN(nn.Module):
```

3. We will then define our `__init__()`:

```
    >>def __init__(self):
        super().__init__()
```

4. Now, we will define our convolutional layers within `__init__()`:

```
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.linear1 = nn.Linear(64 * 4 * 4, 512)
        self.linear2 = nn.Linear(512, 10)
        self.dropout = nn.Dropout(p=0.3)
```

5. Our next step is to write the `forward()` method:

```
    >>def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
```

```
        x = x.view(-1, 64 * 4 * 4)
        x = self.dropout(x)
        x = F.relu(self.linear1(x))
        x = self.dropout(x)
        x = self.linear2(x)
        return x
```

6. Now that our CNN class is complete, we can instantiate our model class:

```
>>model = CNN()
>>model
CNN(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (linear1): Linear(in_features=1024, out_features=512, bias=True)
  (linear2): Linear(in_features=512, out_features=10, bias=True)
  (dropout): Dropout(p=0.3)
)
```

In this recipe, we've completed the model definition.

# How it works...

The way that this recipe works is very similar to what we saw in `Chapter 2`, *Dealing with Neural Networks*, when we looked at a fully connected neural network. We defined a CNN class that is inherited from `nn.Module` in PyTorch by starting with the `__init__()` method and the constructor of the parent class. After that, we defined the various layers in our CNN by passing in the parameters relevant to each layer. For our first convolutional layer, the number of input channels was 3 (RGB), and the number of output channels was defined as 16 and had a square kernel size of 3. The second convolutional layer took in the tensors from the previous layer and had 16 input channels and 32 output channels with a kernel size of 3 x 3. Similarly, the third convolutional layer had 32 input channels and 64 output channels with a 3 x 3 kernel. We also needed a max pooling layer and used a kernel size of 2 and a stride of 2. We used `.view()` to flatten the three dimensions of the tensor into one dimension so that it could be passed into a fully connected network. The -1 in the `view` function ensured that the right size was automatically assigned to that dimension by making sure that the number of elements before and after the `view` function remained the same, which in this case was the batch size.

For the first fully connected layer, we had 1,024 inputs (obtained from flattening the `64 x 4 x 4` tensor after the max pool) and 512 outputs. For the last fully connected layer, we had 512 inputs and 10 outputs, representing the number of output classes. We also defined a dropout layer for our fully connected layer with a probability of 0.3.

Next, we defined the `forward()` method, where we wired together the components defined in the `__init__()` method. So, an input batch of 16 tensors, each with the dimensions of 32 x 32 x 3, went through the first convolutional layer, followed by a ReLU and then a max pooling layer, to form an output tensor with the dimensions of 16 x 16 x 16, and then through the second convolutional layer, followed by a ReLU and a max pool layer, with an output with the dimensions of 8 x 8 x 32, and then through the third convolutional layer, followed by a ReLU and a max pool layer, with the dimensions of 4 x 4 x 64. After this, we flattened the image out to a vector of 1,024 elements and passed it through the dropout layer into the first fully connected layer, giving us 512 outputs, followed by a ReLU and a dropout, into the final fully connected layer to give us the desired number of outputs, which is 10 in our case.

We then instantiated the model from the CNN class and printed the model.

## There's more...

You can experiment with different configurations for dropout, convolution, and pooling layers, and even change the number of each type of layer.

## See also

You can see a different model for training CIFAR10 with CNNs at `https://PyTorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#define-a-convolutional-neural-network`.

# Training an image classifier

Now that we have defined our model, our next major step is to train this model with the data at hand. This is going to be very similar to the training we did in `Chapter 2`, *Dealing with Neural Networks*, with our fully connected neural network. In this recipe, we will finish training our image classifier. It would be really useful if you looked at the *Implementing optimizers* recipe in `Chapter 2`, *Dealing with Neural Networks*, before completing this recipe

# How to do it...

Let's complete the training of our model by going through this recipe:

1. First, we will import `torch` and `torch.optim`:

```
>>import torch
>>import torch.nn as nn
>>import torch.optim as optim
```

2. We will then check for the device that we need to run the model:

```
>>device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
>>device.type
'cuda'
```

3. Then, we will move the model to the available device:

```
>>model = model.to(device)
```

4. Next, we add the cross-entropy loss:

```
>>criterion = nn.CrossEntropyLoss()
```

5. Then, we add the optimizer:

```
>>optimizer = optim.SGD(model.parameters(), lr=0.01)
```

6. Now, we will start the training loop by setting the number of epochs:

```
>>n_epochs = 30
>>for epoch in range(1, n_epochs+1):
    train_loss = 0.0
    valid_loss = 0.0
```

7. Then, we set the model in train mode in the loop:

```
    model.train()
```

8. Then, we loop through each batch:

```
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
```

9. Then, we pass data to the model in the loop:

```
        output = model(data)
```

10. Next, we get the loss:

```
loss = criterion(output, target)
```

11. Then, we back propagate:

```
loss.backward()
```

12. Next, we update the model parameters:

```
optimizer.step()
```

13. Then, we update the total loss:

```
train_loss += loss.item()*data.size(0)
```

14. We then switch the model into evaluation mode, exiting the training batch loop:

```
model.eval()
```

15. Then, we iterate through the validation set batches:

```
for batch_idx, (data, target) in enumerate(valid_loader):
    data, target = data.to(device), target.to(device)
```

16. Next, we get the model output and losses, as we did in *step 9*, *step 10*, and *step 13*:

```
output = model(data)
loss = criterion(output, target)
valid_loss += loss.item()*data.size(0)
```

17. We will then calculate the loss per epoch:

```
train_loss = train_loss/len(train_loader.sampler)
valid_loss = valid_loss/len(valid_loader.sampler)
```

18. Finally, we print the model performance in each epoch:

```
    print(f'| Epoch: {epoch:02} | Train Loss: {train_loss:.3f} |
Val. Loss: {valid_loss:.3f} |')
```

19. Here is a sample of the output that you will see:

```
| Epoch: 01 | Train Loss: 2.027 | Val. Loss: 1.784 |
| Epoch: 02 | Train Loss: 1.640 | Val. Loss: 1.507 |
| Epoch: 03 | Train Loss: 1.483 | Val. Loss: 1.383 |
| Epoch: 04 | Train Loss: 1.380 | Val. Loss: 1.284 |
| Epoch: 05 | Train Loss: 1.312 | Val. Loss: 1.235 |
| Epoch: 06 | Train Loss: 1.251 | Val. Loss: 1.170 |
| Epoch: 07 | Train Loss: 1.198 | Val. Loss: 1.144 |
```

```
| Epoch: 08 | Train Loss: 1.162 | Val. Loss: 1.090 |
| Epoch: 09 | Train Loss: 1.123 | Val. Loss: 1.047 |
| Epoch: 10 | Train Loss: 1.088 | Val. Loss: 1.075 |
| Epoch: 11 | Train Loss: 1.061 | Val. Loss: 1.010 |
| Epoch: 12 | Train Loss: 1.035 | Val. Loss: 0.966 |
| Epoch: 13 | Train Loss: 1.012 | Val. Loss: 0.950 |
| Epoch: 14 | Train Loss: 0.991 | Val. Loss: 0.912 |
| Epoch: 15 | Train Loss: 0.971 | Val. Loss: 0.912 |
| Epoch: 16 | Train Loss: 0.946 | Val. Loss: 0.883 |
| Epoch: 17 | Train Loss: 0.931 | Val. Loss: 0.906 |
| Epoch: 18 | Train Loss: 0.913 | Val. Loss: 0.869 |
| Epoch: 19 | Train Loss: 0.896 | Val. Loss: 0.840 |
| Epoch: 20 | Train Loss: 0.885 | Val. Loss: 0.847 |
| Epoch: 21 | Train Loss: 0.873 | Val. Loss: 0.809 |
| Epoch: 22 | Train Loss: 0.855 | Val. Loss: 0.835 |
| Epoch: 23 | Train Loss: 0.847 | Val. Loss: 0.811 |
| Epoch: 24 | Train Loss: 0.834 | Val. Loss: 0.826 |
| Epoch: 25 | Train Loss: 0.823 | Val. Loss: 0.795 |
| Epoch: 26 | Train Loss: 0.810 | Val. Loss: 0.776 |
| Epoch: 27 | Train Loss: 0.800 | Val. Loss: 0.759 |
| Epoch: 28 | Train Loss: 0.795 | Val. Loss: 0.767 |
| Epoch: 29 | Train Loss: 0.786 | Val. Loss: 0.789 |
| Epoch: 30 | Train Loss: 0.773 | Val. Loss: 0.754 |
```

With this recipe, we have finished training the image classifier.

# How it works...

In this recipe, we found and trained our model. For this, we made our imports and started by identifying and assigning our model to the appropriate device that we have on our machine. We used the `model.to(device)` method to move our model, which is more elegant than using `model.cuda()` or `model.cpu()`.

We then defined our loss function, also called `criterion`. Since this is a classification problem, we used cross-entropy loss. Then, we chose the SGD optimizer to update our model weights on backpropagation, with a learning rate of 0.01, and passed in the model parameters using `model.parameters()`. We then ran our model for 30 epochs, though we could have chosen any reasonable number to do this. In the loop, we reset the training and validation losses to 0 and set the model in train mode, and then we iterated over each batch in the training dataset. We moved the batch first to the device so that, if we had limited GPU memory, not all the data would not be loaded fully into GPU memory. Then, we passed the input tensor into the model and fetched the output and passed it into the loss function to evaluate the difference in the labels that were predicted and the true labels.

After this, we performed backpropagation using `loss.backward()` and updated the model weights using the `optimizer.step()` steps. We then aggregated the loss in the batch using total epoch loss. We then switched the model into the evaluations model using `model.eval()`, since the model's performance needed to be evaluated on the validation set and the model doesn't learn during this phase, and we needed to shut down the dropouts as well. Iterating over the validation batches, we got the model output and accumulated the losses across the validation batches in the entire epoch. After this, we formatted the model performance to see the model changes in each epoch. We noticed that the model training and validation losses decrease over the epochs, which is an indicator that the model is learning.

# There's more...

We have run a trained model and we need to evaluate the model on the holdout data, or the test data, which is the data that the model hasn't seen yet. By doing this, we can evaluate the true performance of the model. For this, you will have to pass into the model test batches, and for each batch, you will have to perform `_, prediction = torch.max(output, 1)` to convert the softmax probabilities into actual predictions and compare the predictions with the true output label using `prediction.eq(target.data.view_as(prediction))`, where we ensure that the dimensions of the prediction and output tensors are the same. This returns a tensor, which will contain 1 where they match and 0 where they don't. We could use this to calculate the accuracy of the model in each batch and aggregate them over the entire test dataset.

# See also

You can see an example implementation of the testing model at `https://PyTorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#test-the-network-on-the-test-data`.

# 4
# Recurrent Neural Networks for NLP

In this chapter, we will deal with **recurrent neural networks** (**RNNs**), a kind of neural network that specializes in dealing with sequential or time-varying data. With convolutional neural networks, we dealt with data points that are spatially related to one another, where a group of pixel values holds information about an image. But think of a rhythm, which is formed by a series of varying sound signals over a span of time. The data points have a temporal relationship to one another. In a recurrent neural network, connections between neurons form a directed graph on a temporal sequence, exhibiting temporal dynamic behavior. A traditional feed-forward network has no memory of the previous input; however, an RNN uses a memory unit to remember the previous input and therefore processes the current input based on the sequence of inputs so far.

In this chapter, we will go through the following recipes:

- Tokenization
- Creating fields
- Developing a dataset
- Developing iterators
- Exploring word embeddings
- Building an LSTM network
- Multilayer LSTMs
- Bidirectional LSTMs

# Introducing RNNs

Here is a diagrammatic representation of an RNN:

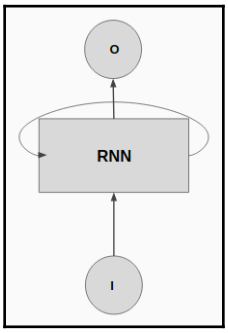



Figure 1: A recurrent neural network

In the preceding diagram, we can see an input, output, and a loop going back on itself in the RNN. RNNs are designed for the persistence of information, and the loop component enables this.

Here is an expanded version of *Figure 1*:

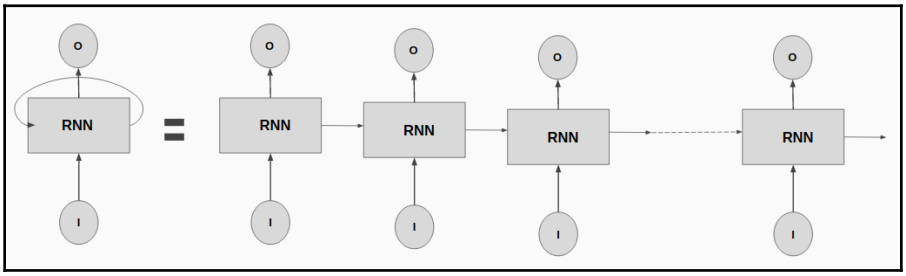



Figure 2: Expanded view of an RNN

In the preceding diagram, we can see an expanded view of the RNN, where information from one step is being fed into the next, creating multiple copies of the same network, and all this is encapsulated in the recurrent loop. A recurrent neural network accepts an input and gives an output, but this output is dependent not just on the input at the given instance, but on the entire history of inputs given to the network, which are mathematically remembered by the network.

A recurrent neural network is capable of taking in an input sequence of a variable size and produce a variably sized output sequence, thereby performing a variable number of computations as opposed to a fixed number of computations in a fully connected neural network. Further RNNs allows for information persistence, and share that information across the inputs that are received. The output that is generated at a given instance is based on the history of all the inputs that it has seen so far.

Here is a diagrammatic representation of an LSTM:

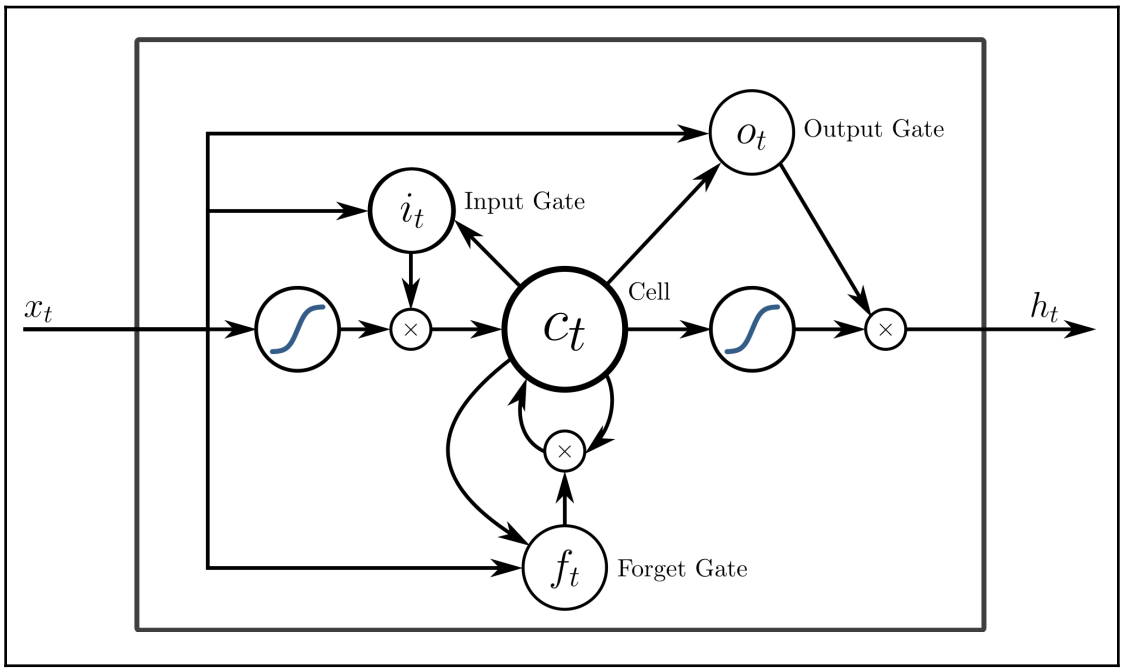



Figure 3: LSTM

**Long short-term memory** (**LSTM**) networks are a type of recurrent neural network that is an advancement on top of RNNs.

Here is a bidirectional LSTM:

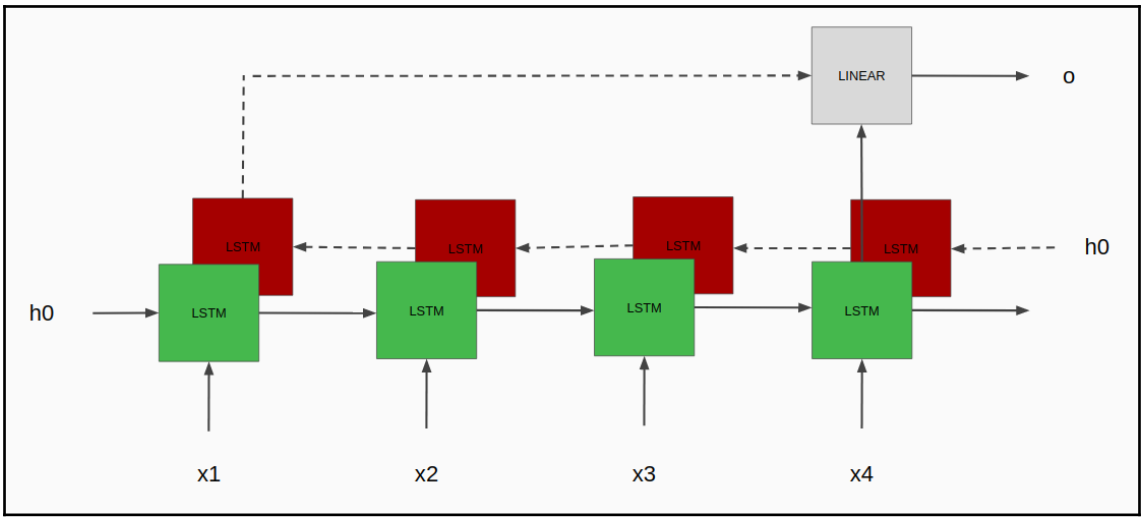



Figure 4: Bidirectional LSTM

Here is a multilayer LSTM:

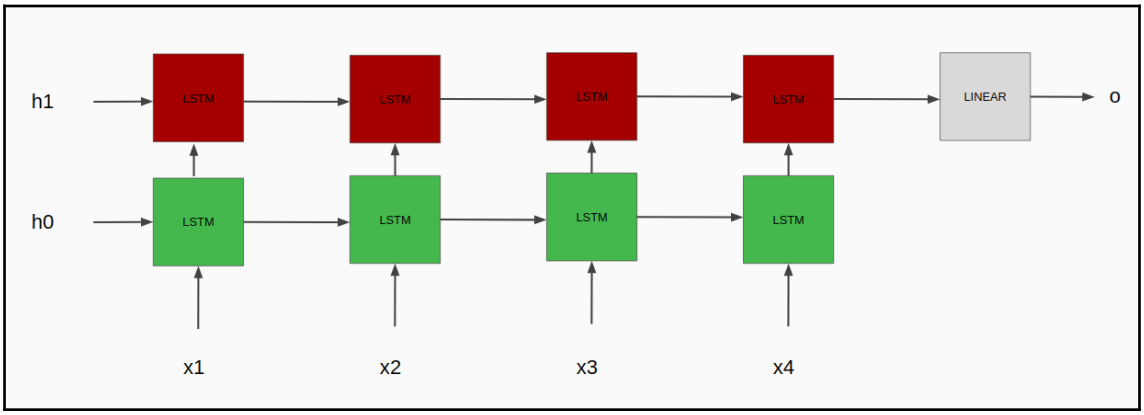



Figure 5: Multilayer LSTM

Bidirectional LSTMs and multilayer LSTMs are improvements over the basic LSTM network architecture.

# Technical requirements

In this chapter, we need to have PyTorch set up. We will be using TorchText, which is a specialized library for dealing with language tasks that work in consortium with PyTorch.

We can install `torchtext` using the following pip command:

```
pip install torchtext
```

With this, we have completed the setup needed for this chapter.

# Tokenization

When dealing with a natural language processing task, we take a text corpus and break it down into smaller units. In this recipe, we will break the sentences down into individual words, where each word represents a meaning along with the other words in its proximity to convey the intent of a sentence. A computer can only understand numbers, and so these words are assigned a unique integer value to represent a word. The process of breaking a sentence into tokens is called tokenization. In this recipe, we will perform word tokenization.

# How to do it...

In this recipe, we will write a tokenizer that we will use in the *Creating fields* section of this chapter:

1. We will first write a simple lambda function:

```
>>tokenizer = lambda words: words.split()
```

2. Then, we will test the `tokenizer()` function:

```
>>tokenizer("This is a test for tokenizer")
['This', 'is', 'a', 'test', 'for', 'tokenizer']
```

In this recipe, we successfully implemented word tokenization.

## How it works...

In this recipe, we wrote a simple tokenizer lambda function that can be used for English and English-like languages. We tokenized the sentences using the spaces between the words. We then tested the tokenizer by passing in a sentence to the `tokenizer()` function. We will use this tokenizer in the next recipe to create fields.

## There's more...

We could also use the `nltk` library for the tokenization of sentences:

```
>>from nltk.tokenize import word_tokenize
>>word_tokenize("This is a test for tokenizer")
['This', 'is', 'a', 'test', 'for', 'tokenizer']
```

Furthermore, there are other types of tokenization, such as string tokenization, which involves tokenizing a string into substrings.

## See also

Various tokenizations of a string using `nltk` can be explored at `https://www.nltk.org/api/nltk.tokenize.html`.

# Creating fields

In this recipe, we will explore fields, which, like the utilities available in TorchVision, make it easy to process natural-language data. Fields let us define the datatype and help us create tensors out of textual data by specifying the set of operations to be performed on the data. The `Field` class lets us perform common text processing tasks and holds the vocabulary of the data at hand.

In this recipe, we will look at how to define various text processing tasks using the fields class.

# How to do it...

In this recipe, we will explore various examples of using fields:

1. We will start with the import:

```
>>from torchtext.data import Field
```

2. For sentiment analysis, we will define a `Field` object for reviews:

```
>>Review = Field(sequential=True, tokenize=tokenizer, lower=True)
```

3. We then define the field for labels:

```
>>Label = Field(sequential=False, use_vocab=False)
```

4. We can add a token at the beginning and end of an input string:

```
>>SequenceField = Field(tokenize=tokenizer, init_token='<sos>',
eos_token='<eos>', lower=True)
```

5. We can set the sequence to a fixed length:

```
>>SequenceField = Field(tokenize=tokenizer, init_token='<sos>',
eos_token='<eos>', lower=True, fix_length=50)
```

6. We can set an unknown token:

```
>>SequenceField = Field(tokenize=tokenizer, init_token='<sos>',
eos_token='<eos>', unk_token='<unk>')
```

7. We can set the batch dimension as the first dimension:

```
>>SequenceField = Field(tokenize=tokenizer, init_token='<sos>',
eos_token='<eos>', unk_token='<unk>', batch_first=True)
```

With this recipe, we've explored the different methods we can use to create fields in TorchText.

# How it works...

In this recipe, we used the `field` class to perform various text processing tasks on a given input text, based on the specific task at hand. In the example for review classification, in the `review` field, we set the `sequential` parameter to `True` since it is sequential data. We would set it to `False` for label fields since they are not sequential. We can set the text to lowercase so that the same word is not assigned separate token IDs based on the case of the tokens. In the case of review classification, this doesn't affect the meaning; this is achieved by setting `lower` to `True`.

For numerical fields, we set `use_vocab` to `False`, which we did for the labels for reviews, as we assumed that the labels had the values `0` for negative and `1` for positive. We passed the `tokenizer` function from the tokenization section as the `tokenize` parameter; we could even use spacy's tokenizer by setting `tokenize="spacy"`. For certain tasks—for instance, using `Sequence` to sequence models—we may need special tokens to indicate the start and end of a sequence. This can be done easily by setting the `init_token` and `eos_token` parameters. This is true for sequence-to-sequence models, and if a token during the model evaluation is not present in the vocabulary that was used to train the model (out of vocabulary), then a custom token can be used to replace these tokens by setting the `unk_token` parameter.

Then, we set the `batch_first` to `True` so that the first dimension of the output tensor is the batch dimension, and if the `fix_length` parameter is set to an integer value, then we would set a fixed length to the input using this field.

# There's more...

We could set the language for tokenization for tokenizations specific to a language, which supports the language supported by spacy. We could set a custom pad token using the `pad_token` parameter. We could define the processing pipelines that will be applied to examples using this field after tokenizing but before numericalizing, and we could do the same after numericalizing but before the numbers are turned into a tensor using the `preprocessing` and `postprocessing` parameters. The `stop_words` parameter can be used to remove tokens that need to be removed while preprocessing. In addition, there is a field type that is specifically available for label fields, called `LabelField`, that can be used instead of normal fields.

## See also

You can learn more about fields at `https://torchtext.readthedocs.io/en/latest/data.html#field`.

# Developing a dataset

In this recipe, we will look at reading text data and using various sources of data. TorchText can read data from text files, CSV/TSV files, JSON files, and directories and converts them into a dataset. Datasets are preprocessed blocks of data that are read into memory, and can be used by other data structures.

## Getting ready

We will use the news classification dataset for this recipe, which you can download from `https://github.com/jibinmathew69/PyTorch1.0-Tutorial/tree/master/NewsClassification`.

It has the following columns in the `.csv` file:

- `id`
- `content`
- `Business`
- `SciTech`
- `Sports`
- `World`

## How to do it...

In this recipe, we will read the toxic comments dataset that is stored as a set of `.csv` files:

1. We will start with the import:

```
>>from torchtext.data import TabularDataset
```

2. We will select the training columns:

```
>>train_datafields = [("id", None),
                     ("content", Review), ("Business", Label),
                     ("SciTech", Label), ("Sports", Label),
                     ("World", Label)]
```

3. Then, we will select the testing columns:

```
>>test_datafields = [("id", None),
                    ("content", Review)]
```

4. Then, we will read the training and validation `.csv` file:

```
>>train, valid = TabularDataset.splits(path='NewsClassification',
                                      train='train.csv',
                                      valid='valid.csv',
                                      format='csv',
                                      skip_header=True,
                                      fields=train_datafields)
```

5. Next, we will read the testing `.csv` file:

```
>>test = TabularDataset(path="NewsClassification/test.csv",
                       format='csv',
                       skip_header=True,
                       fields=test_datafields)
```

6. We will then build the vocabulary:

```
>>Review.build_vocab(train, min_freq=2)
```

With this recipe, we have defined the format of the dataset.

# How it works...

We used the `TabularDataset` module in `torchtext` to read the CSV file, which can also be used to read inputs in the TSV, JSON, and Python dictionaries, which define a dataset of columns. We then defined an array of tuples, where each tuple is a pair of the column and the `Field` object (which defines the textual transformations that are to be applied), but we didn't need a certain column in our final dataset. Then, we set the corresponding `Field` object of that column as `None`, which we saw in the ID column.

In this recipe, we used the news classification dataset. We have one text column to which we applied the `Review` field, and we applied the `Label` field to the rest of the columns. For `test_datafield`, we would have the news content, so for the `content` column, we applied the `Review` field and completely removed the `id` column. We then used the `splits` method in `TabularDataset` and passed in the root folder path where the training and validation files were. We also passed in the filenames of the training and validation files using the `train` and `valid` parameters.

We specified the file format as `csv` and removed the header row by setting `skip_header` to `True`, along with the required columns in the `fields` parameter, and we did the same for testing the dataset. Finally, we called the `build_vocab()` method in the `Fields` object to build the possible library of words with a minimum presence of two times in the dataset. A word that is not in the vocabulary would be assigned an unknown tag in the validation and test sets.

# There's more...

You can use the `Vocab` module to build vocabulary in TorchText. There are other types of dataset other than `TabularDataset`, which can be used depending on the NLP task at hand—for instance, for a language translation task, we could use the `TranslationDataset` class.

# See also

You can read more about the dataset at `https://torchtext.readthedocs.io/en/latest/data.html#torchtext-data`.

# Developing iterators

Iterators are used to load batches of data from the dataset. They provide methods to make loading data and moving data to the appropriate device easier. We could use these iterator objects to iterate over the data while running through the epochs. In this recipe, we will develop these iterators from the dataset. You will need to complete the steps in the *Developing the dataset* recipe, as we will be using the `Dataset` objects from that recipe here.

# How to do it...

In this recipe, we will convert the dataset into iterators so that we have the appropriate batches ready to iterate in each epoch:

1. We will start with the import:

```
>>from torchtext.data import BucketIterator
>>import torch
```

2. We will then define the batch size:

```
>>BATCH_SIZE = 128
```

3. We will then identify the device that's available:

```
>>device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
```

4. Next, we will use `BucketIterator` to create buckets of datasets:

```
>>train_iter, valid_iter, test_iter = BucketIterator.splits(
                                      (train, valid, test),
                                     batch_size=BATCH_SIZE,
                                     device=device,
                                     sort_key=lambda x:
len(x.comment_text),
                                     sort_within_batch=False)
```

With this recipe, we have created our iterators for the training, testing, and validation datasets.

# How it works...

We used iterators to build training, testing, and validations batches and moved the datasets into an appropriate CPU or GPU device. The `Iterators` make it super elegant to do these tasks. We used a specialized iterator class called `BucketIterator`, which groups the input sequences into sequences of similar length and shuffles them automatically. We defined the batch size and found the device that was available on the machine.

We then used the `splits` method of the `BucketIterator` to create training, testing, and validation iterators. We set the `sort_within_batch` parameter to `False`, which is set to `True` if we use the `pack_padded_sequence`, which prevents LSTM from seeing the padded portion of the input sequence. When `True` uses the `sort_key` parameter, it sorts the sequences within the batch in decreasing order.

## There's more...

There are other types of iterators available. A simple iterator loads batches of data from the `Dataset` object, while `BPTTIterator` defines the iterator for language-modeling tasks, with a pair of sequences that are one time-step apart from each other.

## See also

You can find out more about the parameters of iterators at `https://torchtext.readthedocs.io/en/latest/data.html?highlight=bucketiter#iterators`.

# Exploring word embeddings

Word embeddings are learned representations of words. They are dense representations of words, where each word is assigned a vector, that is, a real-valued vector in a pre-defined vector space, rather than a numerical identifier. For instance, a word would be represented as an $[x1, x2, x3, \ldots, xn]$ $n$-dimensional vector—for instance, the word *book* in a corpus might be represented as [0.22, 0.242, ...., 1.234] rather than a one-hot representation of [0, 0, 1, .... , 0] .

A numerical representation is just a representation of a word; however, a word embedding is a representation of a token in which the representation also holds the meaning of the token/word. This meaning is learned by the model from the context in which the word appears. In word embedding, words with similar meanings have a similar representation, and we can perform vector arithmetic on these word vectors, like so:

$$\overrightarrow{king} - \overrightarrow{man} + \overrightarrow{woman} \approx \overrightarrow{queen}$$

Here, we are able to subtract the *man* vector from the *king* vector and sum it with the *woman* vector, and the resulting vector will be close to the vector representation of *queen*. We will explore this implementation in this recipe.

# How to do it...

In this recipe, we will use pretrained embedding with TorchText:

1. We will start with the import:

```
>>from torchtext import vocab
```

2. Then, we will move on to loading the embedding vectors:

```
>>vec = vocab.Vectors('glove.6B.100d.txt',
cache='./vec/glove_embedding/',
url='http://nlp.stanford.edu/data/glove.6B.zip')
```

3. We can build the vocabulary from the pretrained vector by applying it to the field object:

```
>>Review.build_vocab(train, min_freq=2, vectors=vec)
```

With this recipe, we have loaded the pretrained word embedding.

# How it works...

TorchText has a `vocab` module that deals with embeddings. We can download pretrained embeddings by mentioning the name of the embedding that we need in this recipe. We used a pretrained GloVe (a GloVe is a word vector technique) model, that is trained using 6 billion tokens with a 100-embedding dimension vector—glove.6B.50d.

We then loaded the vector from a cache location. If the required embedding is not in the cache, then it is automatically downloaded from the URL and passed as the embedding vector. We then built the vocabulary from those pretrained embeddings, which added to the vocabulary of our training data, using the `build_vocab` method of the `Review` field object.

# There's more...

We can also use the pretrained embedding vocabulary from our training data—for instance, we could use the embedding that's created using gensim for the embedding vectors. We can also create embeddings using the `torch.nn` module; we will see how to do this in the next recipe.

## See also

You can read more about embeddings at `https://torchtext.readthedocs.io/en/latest/vocab.html?highlight=embedding#module-torchtext.vocab`.

# Building an LSTM network

**Long short-term memory** (**LSTM**) networks are a type of recurrent neural network that has internal gates that helps in better information persistence. These gates are tiny neural networks that control when information needs to be saved and when it can be erased or forgotten. RNNs suffer from vanishing and exploding gradients, making it difficult to learn long-term dependencies. LSTMs are resistant to exploding and vanishing gradients, although it is still mathematically possible.

## How to do it...

In this recipe, we will define the LSTM classifier:

1. We will start with the import:

```
>>import torch.nn as nn
```

2. We will name the class `LSTMClassifier`:

```
>>class LSTMClassifier(nn.Module):
```

3. We then add the embedding layer:

```
>>def __init__(self, embedding_dim, hidden_dim, output_dim,
dropout):
            super().__init__()
            self.embedding = nn.Embedding(len(Review.vocab),
embedding_dim)
```

4. Then, we add the LSTM layer:

```
            self.rnn = nn.LSTM(embedding_dim, hidden_dim)
```

5. Then, we add a fully connected layer:

```
            self.fc = nn.Linear(hidden_dim, output_dim)
```

6. Next, we define the dropout layer:

```
self.dropout = nn.Dropout(dropout)
```

7. Then, we define the forward method for the LSTM classifier:

```
>>def forward(self, x):
```

8. Next, we input the embedding layer:

```
x = self.embedding(x)
```

9. Then, we pass the embedding layer output into the LSTM:

```
output, (hidden, cell) = self.rnn(x)
```

10. Then, we apply `dropout`:

```
hidden = self.dropout(hidden)
```

11. Finally, we passed the output through linear layer:

```
return self.fc(hidden)
```

12. We will define the hyperparameters as follows:

```
>>EMBEDDING_DIM = 100
>>HIDDEN_DIM = 256
>>OUTPUT_DIM = 1
>>DROPOUT = 0.5
```

13. Lastly, we create a `model` object:

```
>>model = LSTMClassifier(EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM,
DROPOUT)
```

With this recipe, we have created an LSTM model.

# How it works...

We used the `torch.nn` module to create our model class, `LSTMClassifier`, which is inherited from `torch.nn.Module`, and initialized the base class constructor. We then defined the embedding layer, where the input dimension is the same as the vocabulary size and the output is the embedding dimension, and we then passed the embedding layer output into the LSTM layer, where the input dimension is the embedding dimension, and we defined the hidden state dimension.

We then defined the fully connected layer and the dropout layer. Next, we defined the `forward()` method, which takes in the input sequence, and passed it through the embedding layer, producing an output of dimension `embedding_dim`, which is an embedding vector for the input sequence. This word vector was then passed into the LSTM layer, which outputted three states—the output state, hidden state, and cell state.

The hidden state tensor holds information about all the sequences that the LSTM has seen so far, and so we took the hidden state, applied `dropout`, and passed it through the fully connected layer for the final output vector with a size equal to the number of classes. For instance, for the toxic comment dataset, the number of output classes would be six; however, for a sentiment analyzer with two states–positive and negative–we could even consider having just one output so that `1` represents a positive sentiment and `0` represents a negative sentiment.

# There's more...

For the toxic review task with more than two states, we would use `CrossEntropyLoss()`, and for the sentiment analyzer with just one output, we would use `BCEWithLogitsLoss()`. The rest of the training is the same as what we saw in `Chapter 3`, *Convolutional Neural Networks for Computer Vision*, where we trained a convolutional neural network.

# See also

You can read more about LSTMs at `https://pytorch.org/docs/stable/nn.html#lstm`.

You can read more about vanishing and exploding gradients at `https://www.jefkine.com/general/2018/05/21/2018-05-21-vanishing-and-exploding-gradient-problems/`.

# Multilayer LSTMs

We looked at simple LSTMs in the previous recipe. In this recipe, we will upgrade that simple LSTM definition for multilayer LSTMs. You will need to complete the *Building a LSTM network* recipe to understand this recipe.

# How to do it...

This recipe is a modification that builds on the LSTM recipe.

1. First, we will update the `__init__()` of the class:

```
>>def __init__(self, embedding_dim, hidden_dim, output_dim,
dropout, num_layers):
```

2. We will then add the `num_layers` parameter to the LSTM definition:

```
self.rnn = nn.LSTM(embedding_dim, hidden_dim,
num_layers=num_layers)
```

Our class definition should look as follows:

```
class MultiLSTMClassifier(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, output_dim,
dropout, num_layers):
        self.embedding = nn.Embedding(len(Review.vocab),
embedding_dim)
        self.rnn = nn.LSTM(embedding_dim, hidden_dim,
num_layers=num_layers)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.dropout = nn.Dropout(dropout)
    def forward(self, x):
        x = self.embedding(x)
        output, (hidden, cell) = self.rnn(x)
        hidden = self.dropout(hidden)
        return self.fc(hidden[-1])
```

3. Next, we add a number of layers to the hyperparameters:

```
>>NUM_LAYERS = 2
```

4. Lastly, we create the model object:

```
>>model = MultiLSTMClassifier(EMBEDDING_DIM, HIDDEN_DIM,
OUTPUT_DIM, DROPOUT, NUM_LAYERS)
```

With this recipe, we have modified our network for multilayer LSTMs.

## How it works...

In this recipe, we added `num_layers` and the parameter in the constructor to control the number of layers of LSTMs in the model, and passed it as a keyword argument, `num_layers`, in the LSTM definition.

Then, in the `forward()` method, we took the hidden state only from the last LSTM layer using `hidden[-1]` since the shape of the hidden state is `[num_layers * num_directions, batch, hidden_dim]`, where `num_direction` is `1` by default. This meant that `hidden[-1]` gave the last layer's hidden state. By doing this, we could choose `num_layers` as a hyperparameter. The hidden state output from the lower layer was passed as the input of the higher state.

## There's more...

In this recipe, we only considered the hidden state from the last LSTM layer; however, there can be a complex architecture where all of the hidden layers are used. There is a dropout parameter that can be used to apply dropout in-between the layers of a multilayer LSTM.

## See also

You can read more about multilayer LSTMs at `https://pytorch.org/docs/stable/nn.html#lstm`.

# Bidirectional LSTMs

This recipe builds on the multilayer LSTM recipe. In a normal LSTM, the LSTM reads the input sequence from first to last; however, in a bidirectional LSTM, there is a second LSTM that reads the sequence from last to first—that is, a backward RNN. This type of LSTM improves the model performance when the prediction at the current timestamp is dependent on the inputs further on in the sequence. Consider the examples "I read comics" and "I read comics yesterday". In this case, the same token, that is, *read*, has different meanings based on the token that appears in the future. We will explore its implementation in this recipe.

# Getting ready

This recipe builds on the *Multilayer LSTMs* recipe, and so it is important that you complete that recipe before attempting this one.

# How to do it...

In this recipe, we will modify the class definition from the *Multilayer LSTMs* recipe to make it a bidirectional LSTM:

1. We will set the `bidirectional` parameter to `True`:

```
self.rnn = nn.LSTM(embedding_dim, hidden_dim,
num_layers=num_layers, bidirectional=True)
```

2. We will then change the input dimension of the fully connected layer:

```
self.fc = nn.Linear(2*hidden_dim, output_dim)
```

3. Then, we update the input to a fully connected layer, as follows:

```
hidden = self.dropout(torch.cat((hidden[-2,:,:],
hidden[-1,:,:]), dim=1))
return self.fc(hidden.squeeze(0))
```

The class definition now looks as follows:

```
class BiLSTMClassifier(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, output_dim,
dropout, num_layers):
        self.embedding = nn.Embedding(len(Review.vocab),
embedding_dim)
        self.rnn = nn.LSTM(embedding_dim, hidden_dim,
num_layers=num_layers, bidirectional=True)
        self.fc = nn.Linear(2*hidden_dim, output_dim)
        self.dropout = nn.Dropout(dropout)
    def forward(self, x):
        x = self.embedding(x)
        output, (hidden, cell) = self.rnn(x)
        hidden = self.dropout(torch.cat((hidden[-2,:,:],
hidden[-1,:,:]), dim=1))
        return self.fc(hidden.squeeze(0))
```

4. Then, we create the model object:

```
>>model = BiLSTMClassifier(EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM,
DROPOUT, NUM_LAYERS)
```

With this recipe, we have modified our network so that it's now bidirectional LSTM.

# How it works...

In this recipe, we set the `bidirectional` flag to `True` in the LSTM definition. We concatenated the hidden states of the forward and backward LSTMs and passed them into the fully connected layer. Because of this, the input dimension of the fully connected layer was doubled to accommodate the forward and backward hidden state tensors.

In the `forward()` method, we concatenated the forward and backward hidden states using `torch.cat()`, and we used the last hidden states of the forward and backward LSTMs. In PyTorch, the hidden states are stacked as `[forward_layer_0, backward_layer_0, forward_layer_1, backward_layer_1, ..., forward_layer_n, backward_layer_n]`, and so the required tensors are `hidden[-2,:,:]`, `hidden[-1,:,:]`. After concatenation, we passed the hidden vector into the fully connected layer after squeezing out the extra dimensions.

# There's more...

We have chosen the last forward and backward hidden states and concatenated them, which is the architecture that we chose. However, we could pick any or all of the hidden states, as per the task at hand.

# See also

You can read more about multilayer LSTMs at `https://pytorch.org/docs/stable/nn.html#lstm`.

# 5
# Transfer Learning and TensorBoard

Transfer learning is an important concept in deep learning that has made it possible for us to use deep learning for various day-to-day tasks. It is a machine learning technique where a model trained for a task is reused to create a new model for a similar task. We take a model trained on a large dataset and transfer its knowledge to a smaller dataset. For computer vision tasks with a **convolutional neural network** (**CNN**), we freeze the early convolutional layers of the network and only train the last few layers. The early convolutional layers extract general, low-level features that are applicable across images for detecting edges, patterns, and gradients, while the later layers identify specific features within an image, and are specific to the dataset.

In this chapter, we will train our image classifier to distinguish between the chest X-rays of normal patients and pneumonia patients, and we will use a trained ResNet-50 model to perform transfer learning. We will replace the classifier and have two output units to represent the normal and pneumonia classes.

We will go through the transfer learning task in the following stages:

1. Load the pretrained ResNet-50 model, trained on an ImageNet dataset.
2. Freeze parameters (weights) in the model's lower convolutional layers.
3. Replace the classifier with multiple layers of trainable parameters.
4. Train classifier layers on the training data available for the task.
5. Fine-tune the hyperparameters and unfreeze more layers as needed.

In this chapter, we will cover the following recipes:

- Adapting a pretrained model
- Implementing model training
- Implementing model testing
- Loading the dataset
- Defining the TensorBoard writer
- Training the model and unfreezing layers

# Technical requirements

To complete this recipe, we need Torch version 1.2 or above, and it is highly recommended that we have a CUDA-enabled device.

# Adapting a pretrained model

In this recipe, we will take a pretrained ResNet model and modify the last layers to suit the output we require. We need only two classes compared to the number of classes in the ImageNet dataset that are used to train the ResNet-50 model. We will modify the last pooling layer and the fully connected classifier of the ResNet model. We will further restrict the training of the model to only the newly added classifier unit, and all the remaining layers will be preserved from updating the weights. This is called freezing the model. Let's look at how to implement the recipe.

## Getting ready

This recipe requires us to download a particular dataset. We will get the dataset from `https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia/download`. In order to complete this recipe, your PyTorch installation should be 1.2 or above, and it is highly recommended that you use a CUDA-enabled device.

# How to do it...

In this recipe, we will train our neural network, and we will start with a pretrained model, ResNet-50, that was trained on an ImageNet dataset:

1. We will now write our Python code, starting with imports:

```
>>import torch
>>import torch.nn as nn
>>import numpy as np
>>import torch.optim as optim
>>from torchvision import transforms, datasets, models, utils
>>import time
>>import numpy as np
>>from torchsummary import summary
>>from torch.utils.data import DataLoader
```

2. Define the `AdaptiveConcatPool2d` submodule:

```
>>class AdaptiveConcatPool2d(nn.Module):
    def __init__(self, sz=None):
        super().__init__()
        sz = sz or (1,1)
        self.ap = nn.AdaptiveAvgPool2d(sz)
        self.mp = nn.AdaptiveMaxPool2d(sz)
    def forward(self, x):
        return torch.cat([self.mp(x), self.ap(x)], 1)
```

3. Define a function to get the model:

```
>>def get_model():
    model = models.resnet50(pretrained=True)
```

4. We will now freeze the model:

```
for param in model.parameters():
    param.requires_grad = False
```

5. We will now replace the last two layers of the ResNet and return the model:

```
model.avgpool = AdaptiveConcatPool2d()
model.fc = nn.Sequential(
    nn.Flatten(),
    nn.BatchNorm1d(4096),
    nn.Dropout(0.5),
    nn.Linear(4096, 512),
    nn.ReLU(),
    nn.BatchNorm1d(512),
```

```
            nn.Dropout(p=0.5),
            nn.Linear(512, 2),
            nn.LogSoftmax(dim=1)
        )
        return model
```

With this recipe, we have our function ready to get the model.

# How it works...

In this recipe, we defined a submodule, `AdaptiveConcatPool2d`, that performs concatenation between Average 2D pooling and Max 2D pooling, so that there is smooth transition from the convolution layers to the fully connected layers with maximum feature information.

We then defined the `get_model()` function, which first downloads the ResNet-50 model (which is not available locally) and freezes the weights of the model. By freezing the weights, the lower convolutional layers are not updated. Then we replaced the average pooling layer with our `AdaptiveConcatPool2d` layer and added a fully connected classifier with two output units for the two classes available. We finally returned the model with the frozen ResNet layers.

# Implementing model training

In this recipe, we will implement a function for training the model in a single epoch. This function further logs the training metrics of the model and plots them on to TensorBoard. We will pass in the model, training data, optimizer, and criterion for model training and it will return the training loss.

# How to do it...

We will implement the training function in this recipe:

1. Define the training function:

```
>>def train(model, device, train_loader, criterion, optimizer,
epoch, writer):
    model.train()
    total_loss = 0
```

2. Iterate over the training data and update the model weights:

```
for batch_id, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)

    optimizer.zero_grad()
    preds = model(data)
    loss = criterion(preds, target)
    loss.backward()
    optimizer.step()
    total_loss += loss.item()
```

3. Log and return the training loss:

```
    writer.add_scalar('Train Loss', total_loss/len(train_loader),
epoch)
    writer.flush()
    return total_loss/len(train_loader)
```

With this recipe, we have completed the training function.

# How it works...

In this recipe, we defined a function to perform a training epoch. We set the model in training mode using `.train()` before starting the training process, and set the training loss to `0`. We then iterated over the training data and moved the input data points and their corresponding labels to the available device (CPU or GPU).

We then cleared the gradient, made the model prediction, and passed it to the criterion to determine the training loss. We then backpropagated the loss and updated the weights of the model. Since the model is frozen, it only updates the weights of the classifier in the model.

We finally logged the training metric—training loss—in TensorBoard using the `add_scalar()` method in the `SummaryWriter` object from TensorBoard, where we passed in a label, a scalar value, and a counter, which in our case is the epoch number.

# Implementing model testing

In this recipe, we will define a function to test the model on validation data in an epoch. This function also logs the testing metrics on to TensorBoard. We will also add utility functions to log some of the misclassifications from the model by plotting the images and labeling these images in a readable manner.

# How to do it...

In this recipe, we will implement model testing along with utility functions:

1. First, we will define a function to convert our tensor to images:

```
>>inv_normalize = transforms.Normalize(
        mean=[-0.485/0.229, -0.456/0.224, -0.406/0.225],
        std=[1/0.229, 1/0.224, 1/0.255]
    )
```

2. Then, we will define a function to log misclassified images:

```
>>def misclassified_images(pred, writer, target, data, output,
epoch, count=10):
    misclassified = (pred != target.data)
    for index, image_tensor in
enumerate(data[misclassified][:count]):
        img_name = '{}->Predict-{}x{}-Actual'.format(
                epoch,
                LABEL[pred[misclassified].tolist()[index]],
                LABEL[target.data[misclassified].tolist()[index]],
            )
        writer.add_image(img_name, inv_normalize(image_tensor),
epoch)
```

3. We will now have a function for and logging the metrics:

```
>>def test(model, device, test_loader, criterion, epoch, writer):
    model.eval()
    total_loss, correct = 0, 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            total_loss += criterion(output, target).item()
            pred = output.data.max(1)[1]
            correct += pred.eq(target.data).cpu().sum()
```

```
                misclassified_images(pred, writer, target, data,
output, epoch)
    total_loss /= len(test_loader)
    accuracy = 100. * correct / len(test_loader.dataset)
```

4. We now log test metrics on TensorBoard:

```
    writer.add_scalar('Test Loss', total_loss, epoch)
    writer.add_scalar('Accuracy', accuracy, epoch)
    writer.flush()
    return total_loss, accuracy
```

With this recipe, we have completed the testing function.

# How it works...

In this recipe, we wrote an inverse normalizing function to undo the normalization that we established while converting the image into a tensor with ImageNet stats. We also defined a `misclassified_images()` method that logs the images where the prediction went wrong. The misclassified images were then added into TensorBoard using the `add_image()` method in the `SummaryWriter` object, which takes in the image name, image, and counter.

Then we defined the `test()` method, which runs a validation on the validation dataset for the model and logs the test loss and accuracy with the `add_scalar()` method, as in the training function. We finally returned the test loss and the model accuracy on the validation dataset.

# Loading the dataset

In this recipe, we will load our pneumonia dataset and convert it into tensor. The model requires data in the form of tensors, and so we will need to perform preprocessing on the image to give it the required data. We will perform data augmentation to increase our dataset size. We will also perform image normalization as per the ImageNet dataset before feeding it into the model.

# How to do it...

In this recipe, we will load the dataset:

1. First, we will define the transforms:

```
>>image_transforms = {
```

The following code shows the training set transforms:

```
    'train':
    transforms.Compose([
        transforms.RandomResizedCrop(size=300, scale=(0.8, 1.1)),
        transforms.RandomRotation(degrees=10),
        transforms.ColorJitter(0.4, 0.4, 0.4),
        transforms.RandomHorizontalFlip(),
        transforms.CenterCrop(size=256), # Image net standards
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225]) # Imagenet
standards
    ]),
```

The following code shows the validation set transforms:

```
    'val':
    transforms.Compose([
        transforms.Resize(size=300),
        transforms.CenterCrop(size=256),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225])
    ]),
```

The following code shows the test set transforms:

```
    'test':
    transforms.Compose([
        transforms.Resize(size=300),
        transforms.CenterCrop(size=256),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225])
    ]),
}
```

2. Then, we will define the image paths, batch size, log path, and path to save the model:

```
>>datadir = '../input/chest-xray-pneumonia/chest_xray/chest_xray/'
>>traindir = datadir + 'train/'
>>validdir = datadir + 'test/'
>>testdir = datadir + 'val/'
>>model_path = "model.pth"
>>batch_size = 128
>>PATH_to_log_dir = 'logdir/'
```

3. Next, let's load the images from the folder:

```
>>data = {
    'train':
    datasets.ImageFolder(root=traindir,
transform=image_transforms['train']),
    'val':
    datasets.ImageFolder(root=validdir,
transform=image_transforms['val']),
    'test':
    datasets.ImageFolder(root=testdir,
transform=image_transforms['test'])
}
```

4. Now, we will create iterators:

```
>>dataloaders = {
    'train': DataLoader(data['train'], batch_size=batch_size,
shuffle=True),
    'val': DataLoader(data['val'], batch_size=batch_size,
shuffle=True),
    'test': DataLoader(data['test'], batch_size=batch_size,
shuffle=True)
}
```

5. Then we will build the class labels:

```
>>LABEL = dict((v,k) for k,v in data['train'].class_to_idx.items())
```

With this recipe, we have our dataset ready.

## How it works...

In this recipe, we defined the transforms that we want for our images in the training, validation, and test datasets. The transforms are selected as per the dataset, and the values in the normalize transform come from the ImageNet stats. We then defined our dataset paths, model name, batch size, and log directory. We then used the `datasets.ImageFolder()` method to load the data according to the folder name and created an iterator for each of the datasets.

Note that we flipped the validation dataset and test dataset directories. This is because the validation dataset for the given dataset is really small, and so we used the test dataset for our validation dataset.

We also used `DataLoader` to create iterators for our training, testing, and validation datasets. Then we created a `LABEL` constant, which is a dictionary that maps the classifier output index to the class names.

# Defining the TensorBoard writer

In this recipe, we will create an object that writes on to TensorBoard. We use the `SummaryWriter` object to write into TensorBoard. We can use TensorBoard to write scalar values, plot graphs, and plot images, among other functionalities. We will define a function that returns a TensorBoard `SummaryWriter` object to log our model metrics.

## Getting ready

This recipe requires us to install the TensorBoard library.

We need to install the TensorBoard nightly version:

```
pip install tb-nightly
```

With this, we are ready to implement the recipe.

# How to do it...

In this recipe, we will create our writer object to log data on to TensorBoard.

1. We will load TensorBoard by typing the following in the command line:

```
tensorboard --logdir=log_dir/ --port 6006
```

   You can access TensorBoard on your browser by going to `http://localhost:6006/`.

2. Next, we will import TensorBoard:

```
>>from torch.utils.TensorBoard import SummaryWriter
```

3. We will then define a function to get the TensorBoard writer:

```
>>def tb_writer():
      timestr = time.strftime("%Y%m%d_%H%M%S")
      writer = SummaryWriter(PATH_to_log_dir + timestr)
      return writer
```

4. Then we will create an image grid to visualize the images in our dataset:

```
>>writer = tb_writer()
>>dataiter = iter(dataloaders['train'])
>>images, labels = dataiter.next()
>>grid = utils.make_grid([inv_normalize(image) for image in
images[:32]])
>>writer.add_image('X-Ray grid', grid, 0)
>>writer.flush()
```

With this, we have made TensorBoard ready.

# How it works...

In this recipe, we started by loading our TensorBoard from the command line to read from the `logdir/` directory. TensorBoard doesn't throw an error even if the directory isn't present but rather waits for the directory to appear, and so it is important to pass the right directory.

We then imported TensorBoard into the code and defined a `tb_writer()` function, which returns a new `SummaryWriter` object. We passed the `writer` object the directory name into which the TensorBoard logs are to be saved, and ensure that each writer has a unique directory that it writes to, with the help of the timestamp, `timestr`.

We then created an image grid to see the sample images in our training dataset using `utils.make_grid()` and passed it to TensorBoard using the `add_image()` method. We fetched the images using `iter(dataloaders['train'])`, then picked a sample of 32 images and performed inverse normalization before making the grid. We also used the `flush()` method in the writer to write the buffer data into storage.

# Training the model and unfreezing layers

In this recipe, we will complete our model training for a predefined number of iterations of the dataset. We will save the best models during the model training. Once the model is trained for the given number of epochs, we will load the model with the weights of the best model. We will then unfreeze the previously frozen ResNet layers of the model and train the model to fine-tune the weights with a lower learning rate.

## How to do it...

In this recipe, we will complete our model training.

1. Move the model to an available device:

```
>>device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
>>model = get_model().to(device)
```

2. Define the criterion and optimizer:

```
>>criterion = nn.NLLLoss()
>>optimizer = optim.Adam(model.parameters())
```

3. We will define a function to train our model over epochs:

```
>>def train_epochs(model, device, dataloaders, criterion,
optimizer,epochs, writer):
    print('{0:>20} | {1:>20} | {2:>20} | {3:>20}
|'.format('Epoch','Training Loss','Test Loss', 'Accuracy'))

    best_score = np.Inf

    for epoch in epochs:

        train_loss = train(model, device, dataloaders['train'],
criterion, optimizer, epoch, writer)

       test_loss, accuracy = test(model, device,
dataloaders['val'], criterion, epoch, writer)

        if test_loss < best_score:
            best_score = test_loss
            torch.save(model.state_dict(), model_path)

        print('{0:>20} | {1:>20} | {2:>20} | {3:>20.2f}%
|'.format(epoch,train_loss,test_loss, accuracy))

        writer.flush()
```

4. We will now train our frozen model:

```
>>train_epochs(model, device, dataloaders, criterion, optimizer,
range(0,10), writer)
>>writer.close()
```

Here is a sample output:

```
Epoch | Training Loss | Test Loss    | Accuracy |
 0    |  0.3617607994 | 0.4970110774 |  79.17%  |
 1    |   0.233799973 | 0.3870731115 |  84.78%  |
 2    |  0.2014380195 |   0.37044851 |  85.26%  |
 3    |   0.190022166 |  0.362625807 |  86.86%  |
 4    |   0.176903085 |   0.40945090 |  85.90%  |
 5    |   0.163670904 | 0.3690894782 |  86.86%  |
 6    |  0.1607481929 |  0.418265098 |  84.46%  |
 7    |  0.1615160162 | 0.4016072392 |  85.58%  |
 8    |  0.1519727988 |  0.481940734 |  84.13%  |
 9    |  0.1441831755 |  0.433110350 |  84.46%  |
```

5. We will define a function to unfreeze the model:

```
>>def unfreeze(model):
    for param in model.parameters():
        param.requires_grad = True
```

6. We will load the best model so far and unfreeze the model:

```
>>model.load_state_dict(torch.load(model_path))
>>unfreeze(model)
```

7. We will update the optimizer for a lower learning rate:

```
>>optimizer = optim.Adam(model.parameters(), lr=1e-6)
```

8. We will train the unfrozen model:

```
>>writer = tb_writer()
>>train_epochs(model, device, dataloaders, criterion, optimizer,
range(9,14), writer)
```

Here is a sample output:

```
Epoch |        Training Loss |            Test Loss |  Accuracy  |
    9 |  0.15972968554351388 |  0.41342413425445557 |   85.10%   |
   10 |   0.1224460500042613 |   0.3801746487617493 |   86.54%   |
   11 |   0.1217333177422605 |  0.37790409922599794 |   87.18%   |
   12 |  0.11098722713749583 |   0.3712982594966888 |   87.98%   |
   13 |  0.09877484373566581 |  0.41088773012161256 |   86.70%   |
   14 |  0.09256085244620718 |   0.3181425631046295 |   89.42%   |
```

9. We will finally close our TensorBoard writer:

```
>>writer.close()
```

In the following screenshot, we can see the plots that were generated from training the model viewed in TensorBoard:

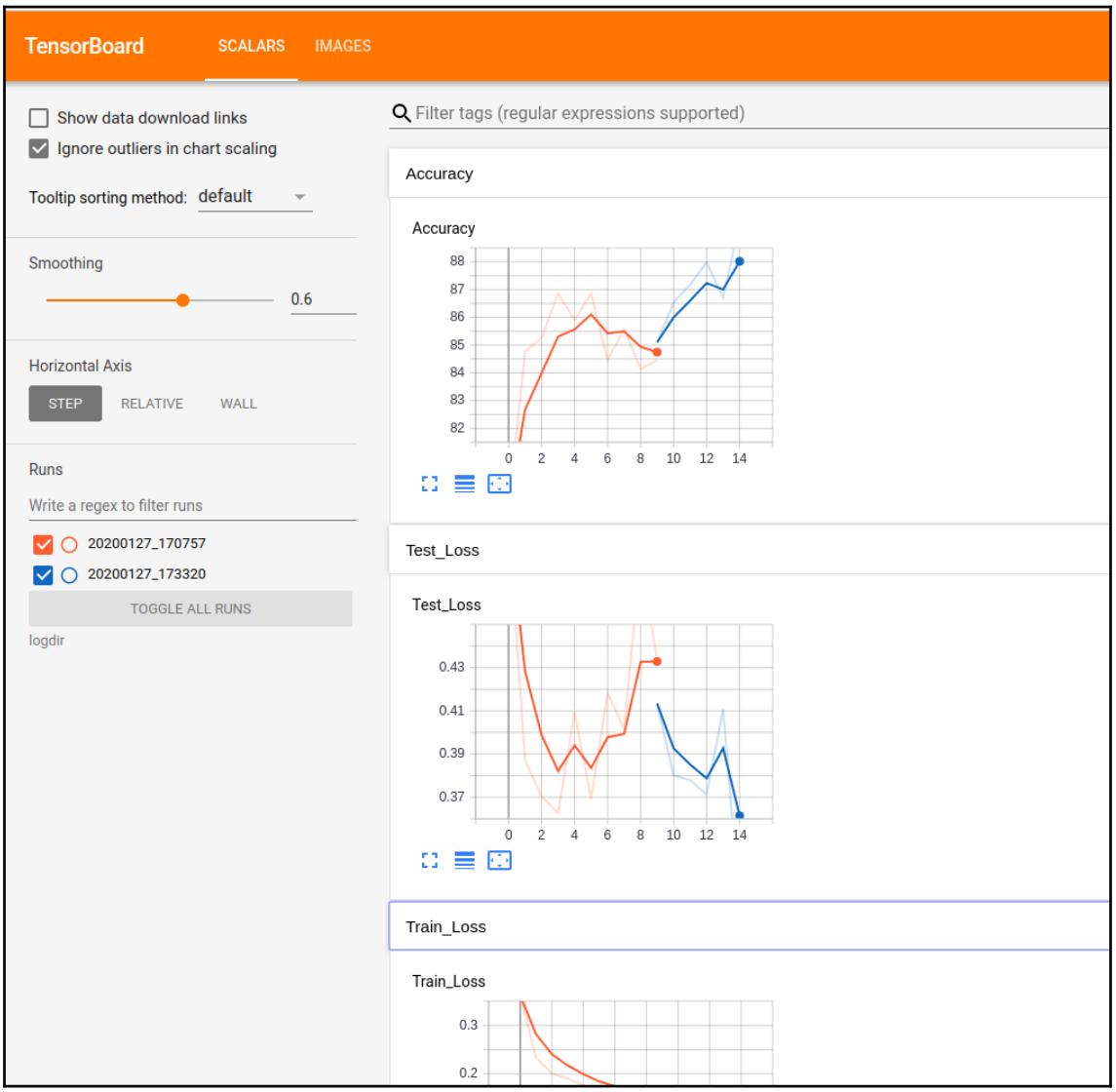



Figure 1: TensorBoard main dashboard

Here is the image grid we created:

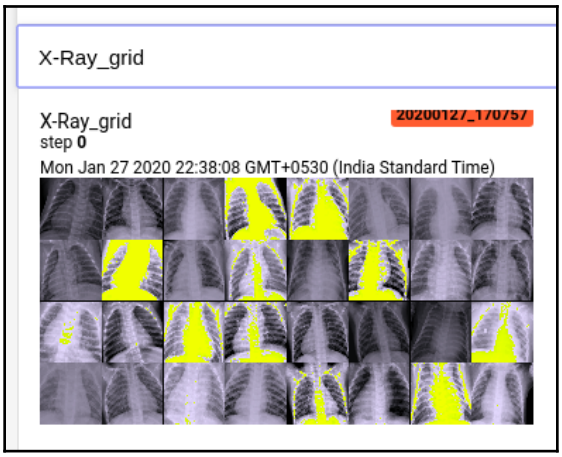



Figure 2: Image grid

Here is a misclassification example that predicted pneumonia, but that is actually normal:

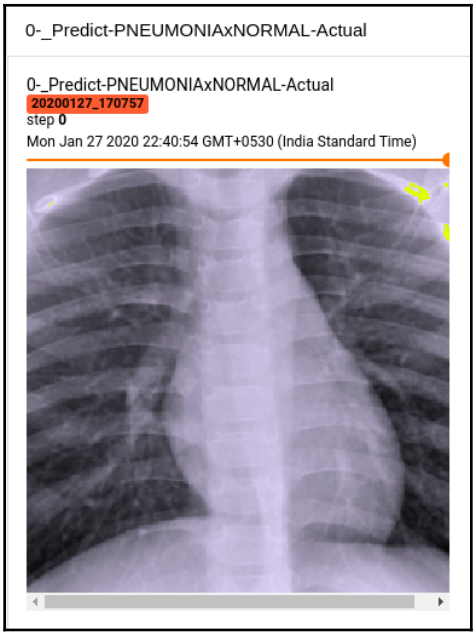



Figure 3: Predicted pneumonia, actually normal

Here is a misclassification example that predicted normal, but that is actually pneumonia:

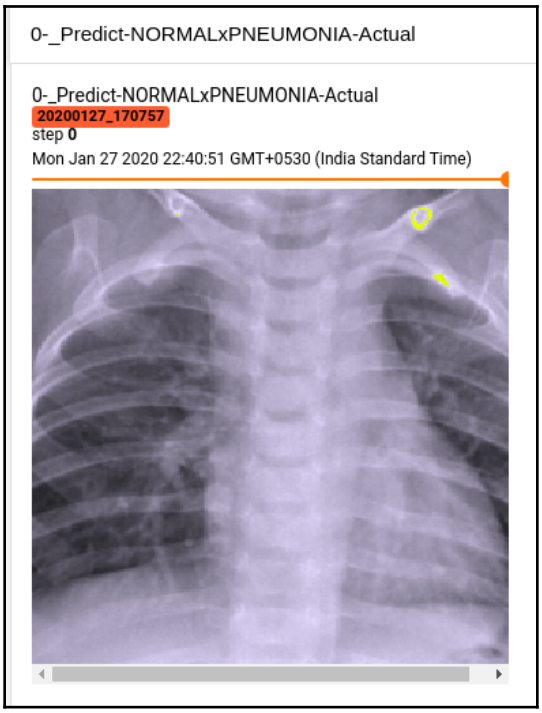



Figure 4: Predicted normal, actually pneumonia

Here is a plot showing our training loss decreasing over epochs:

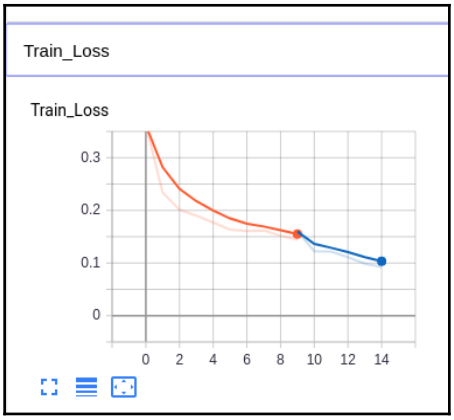



Figure 5: Training loss

Here is a plot showing our test loss decreasing over epochs:


Figure 6: Testing loss

Here is a plot showing our accuracy increasing over epochs:

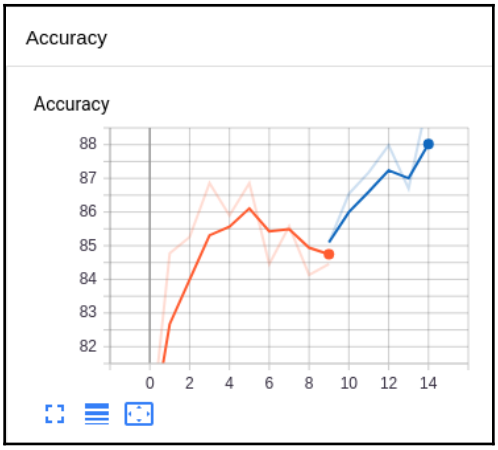



Figure 7: Accuracy score

With this recipe, we have trained our model and visualized its results in TensorBoard.

# How it works...

In this recipe, we created the model and moved it to the available device, and used negative log loss and Adam as our criterion and optimizer respectively. The `train_epochs()` method is used to train the model over a defined range of epochs. At the end of each epoch, we used the `writer.flush()` method to ensure that all pending events have been written to disk. Finally, we used the `writer.close()` to flush close the writer. We also saved the best models in this function to be reloaded later.

We then reloaded the best model from our training so far and unfroze it for fine-tuning. With unfreezing, all the model parameters are available for training. We set the optimizer to a low learning rate, trained this unfrozen model for a few more epochs, and logged the model performance. We saw that our model performs better with fine-tuning.

From the TensorBoard plot, we saw the metrics from the frozen model in orange and the metrics after unfreezing in blue, which indicates the improvement in performance post-model unfreeze. We then had the plots of the image grid sampled from the training data and misclassification examples from various epochs.

We saw that the training and test loss decreased and the accuracy increased over the epochs.

# There's more...

In this recipe, we could further write a function to determine the test dataset metrics, add histograms to our TensorBoard using the `add_histogram()` method, and train the model using other pretrained networks.

# See also

For more details, refer to the following:

- You can read more about fine-tuning at `https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html`.
- You can learn about another transfer learning example at `https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html`.
- You can explore TensorBoard functions at `https://pytorch.org/docs/stable/TensorBoard.html`.

# 6
# Exploring Generative Adversarial Networks

A **generative adversarial network** (**GAN**) is a machine learning technique in which two models are trained simultaneously: one specializes in creating fake data and the other specializes in distinguishing between the fake data and the real data. The term *generative* reflects the fact that these neural networks are used to create new data, and the term *adversarial* comes from the fact that the two models compete against one another, improving the quality of the generated data.

The two models within the GAN are known as the generator and the discriminator, where a generator is responsible for creating data and the discriminator takes in data and classifies it as real or generated by the generator. The goal of the generator is to create data samples that are indistinguishable from real data in the training set.

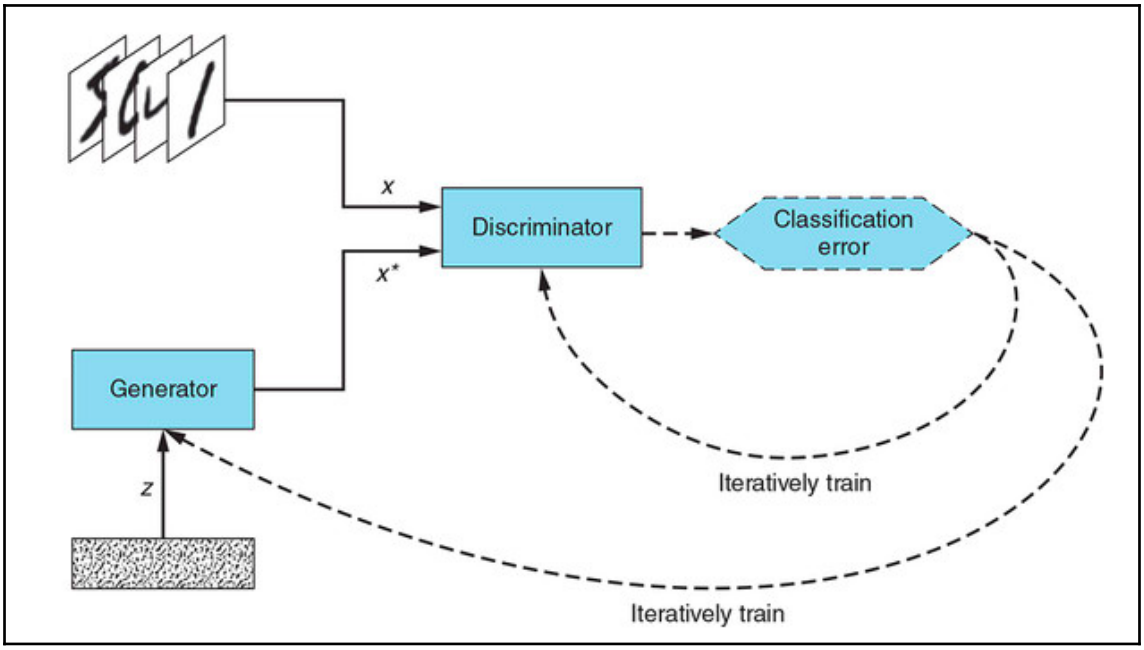
We can understand the concept of the GAN using an analogy where a criminal (the generator) wants to forge money and a detective (the discriminator) is trying to catch him. The more authentic-looking the fake currency becomes, the better and more efficient the detective must be at detecting the fake currency, which in turn means that the quality of the forged bills has to go up enough to remain undetected by the detective.

The generator learns from the discriminator's classification feedback. The discriminator's goal is to determine that its input is real (from the training dataset) or fake (from the generator), and so every time the discriminator makes the mistake of classifying a fake image as real, the generator gets positive feedback that it did a good job. Conversely, each time the discriminator correctly catches a generator-produced image as fake, the generator receives the feedback that it needs to improve.

The discriminator basically is a classifier, and like any classifier, it learns from how far off its predictions are from the true labels, which in this case are either real or fake. So as the generator gets better at producing realistic-looking data, the discriminator has to get better at telling fake from real labels. In this way, both networks improve side by side.

Conceptually, a generator must be able to capture the characteristics of real data from the training examples so that the samples it generates are indistinguishable from the real data. The generator learns to create patterns (instead of recognizing patterns in image classification problems) by itself. Usually, the input to a generator is often a vector of random numbers.

Let's look at the following architectural diagram for a GAN:




In this diagram, there is a data source containing the training image, $x$, whose properties the generator has to capture and recreate. The generator takes in a random vector, $z$, that acts as a seed for the generator to create fake images. The generator takes the seed and generates images, $x^*$, and the discriminator takes in images from the real and fake images and outputs a probability that the given input is real (assuming that real images are represented with 1 and fake images represented with 0). We then obtain the classification error and use it to iteratively train the discriminator and generator. The objective of the discriminator is to minimize the classification error and the objective of the generator is to maximize the classification error.

Theoretically, the generator and discriminator reach an equilibrium, where the generator has captured all the features of the real image in the fake image that it generates, and has nothing to gain from further training. Similarly, the discriminator could only guess that the image is either fake or real with a 50% probability, as both the images are completely indistinguishable from one another in terms of their properties. At that state, GANs are said to have converged; however, practically, such a state is hard to achieve. In this chapter, we will explore the concept of GANs and the implementation of various types of GAN in PyTorch.

In this chapter, we will cover the following recipes:

- Creating a DCGAN generator
- Creating a DCGAN discriminator
- Training DCGAN model
- Visualizing DCGAN results
- Running PGGAN with PyTorch hub

# Technical requirements

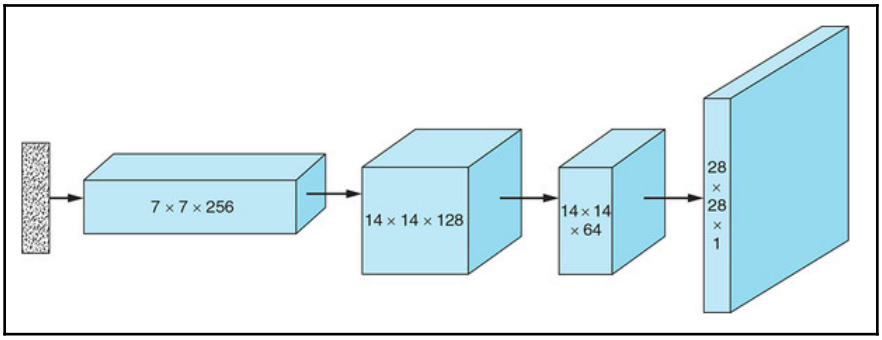
It is highly recommended that, for the recipes implemented in this chapter, the code is run on a machine that has an NVIDIA GPU, and has CUDA and CUDNN enabled, as the recipes in this chapter are computationally intensive.

# Creating a DCGAN generator

In this recipe and the recipes that follow, we will implement a DCGAN. DCGAN stands for Deep Convolutional GAN; they are a significant improvement over vanilla GANs. In DCGANs, we use convolutional neural networks as opposed to fully connected networks in vanilla GANs. In `Chapter 3`, *Convolutional Neural Networks for Computer Vision*, we saw how the fully connected classifier from `Chapter 2`, *Dealing with Neural Networks*, is an improvement in the field; the same is true for DCGANs over vanilla GAN. In DCGAN, we will use batch normalization, which is a technique in which we normalize the output of a layer being fed as an input to the next layer. Batch normalization allows each layer of a network to learn independently of other layers, reducing the covariate shift.

Batch normalization is achieved by scaling so that the mean is 0 and the variance is 1. In this recipe, we will be generating handwritten digits, similar to the MNIST dataset, containing data from a noise vector. We will expand this noise vector, convert it into a 2D matrix, and finally convert it into a 28 x 28 black and white image. In order to increase the height and width, we will have to perform the reverse of the convolution operation—this is called deconvolution. We will do this while performing the classification task using convolution. While performing deconvolutions, we will increase the height and width while reducing the number of channels.

The following is our architecture diagram for our DCGAN generator:




Note that we will be using concepts from `Chapter 3`, *Convolutional Neural Networks for Computer Vision*, and so it would be good to go through those recipes again.

# How to do it...

In this recipe, we will implement the generator side of the GAN network:

1. We will start with imports:

```
>import torch
>>import torch.nn as nn
>>import torchvision.transforms as transforms
```

2. We will then define transforms:

```
>>transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, ), (0.5, )),
])
```

3. Then we will make the device available:

```
>>device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
```

3. Now we will define the generator class:

```
>>class Generator_model(nn.Module):
    def __init__(self, z_dim):
         super().__init__()
```

4. Then we will define the units of the generator:

```
             self.fc = nn.Linear(z_dim, 256 * 7 * 7)
             self.gen = nn.Sequential(
                        nn.ConvTranspose2d(256, 128, 4, 2, 1),
                        nn.BatchNorm2d(128),
                        nn.LeakyReLU(0.01),
                        nn.ConvTranspose2d(128, 64, 3, 1, 1),
                        nn.BatchNorm2d(64),
                        nn.LeakyReLU(0.01),
                        nn.ConvTranspose2d(64, 1, 4, 2, 1),
                        nn.Tanh()
                    )
```

5. Now we will define the forward method:

```
def forward(self, input):
    x = self.fc(input)
    x = x.view(-1, 256, 7, 7)
    return self.gen(x)
```

6. Finally, we will create the object for the generator model:

```
>>generator = Generator_model(z_dim).to(device)
```

Having gone through this process, we have our DCGAN generator ready.

# How it works...

In this recipe, we performed transforms to convert an image into a tensor and normalize it, just as we did in `Chapter 3`, *Convolutional Neural Networks for Computer Vision*. We then identified the device that we have on our machine: CPU or GPU. We then defined our `Generator_model` class that inherits from the `nn.Module` class, just as we did in all our previous architectures.

In the constructor, we passed the `z_dim` parameter, which is our noise vector size. We then defined a fully connected unit, `self.fc`, to which we passed the noise vector, and gave it `256 * 7 * 7` outputs. We then defined a `nn.Sequential` unit called `self.gen`, which held the key components for defining the generator. We used a set of deconvolutions, batch normalization, and activation layers, using `nn.ConvTranspose2d`, `nn.BatchNorm2d`, and `nn.LeakyReLU`, available in PyTorch. `ConvTranspose2d` takes in input channels, output channels, kernel size, stride, and padding, among other parameters. `BatchNorm2d` takes in the number of features/channels from the previous layer as its argument and LeakyReLU takes in the angle of negative slope.

Unlike ReLU, LeakyReLU allows passing a small gradient signal for negative values. It makes gradients from the discriminator flow into the generator. We used tanh activation in the output layer, but from the DCGAN paper we observed that using a bounded activation allowed the model to learn to rapidly saturate and cover the color space of the training distribution. It could be that the symmetry of tanh is an advantage here, since the network should be treating darker colors and lighter colors in a symmetric way.

Let's look at how the `forward` method works. The input noise vector of the `z_dim` dimension goes through the fully connected layer to give a 12544 output. We then reshape the 12544 outputs into `256 x 7 x 7`, where 256 is the number of channels. The `256 x 7 x 7` tensor then goes through the deconvolution layer to give a `128 x 14 x 14` output, and then goes through the batchnorm layer with 128 features and leaky ReLU. The `128 x 14 x 14` is then converted to a `64 x 14 x 14` tensor in the second deconvolution, and in the third deconvolution it becomes a `1 x 28 x 28` tensor; these are just the dimensions we need. We then create the generator object and move it to device.

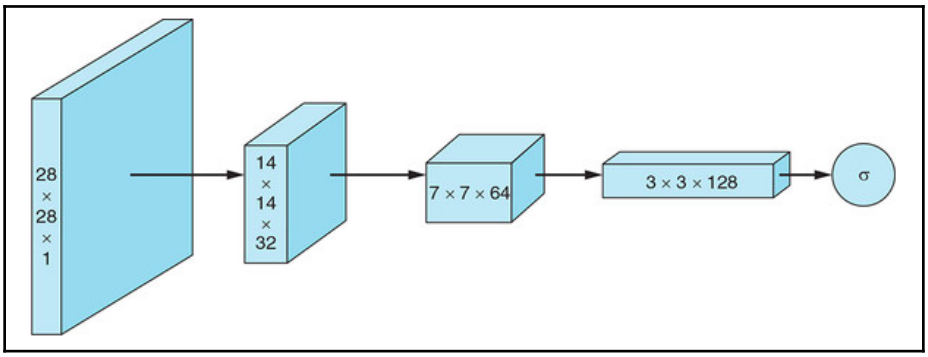
## See also

You can learn more about DCGAN at `https://arxiv.org/pdf/1511.06434.pdf`.

# Creating a DCGAN discriminator

In this recipe, we explore the discriminator side of the GAN network. Basically, the discriminator is a classifier that classifies between two classes—that is, according to whether the given image is a real image from the dataset or a fake image generated by the generator network. It is from the feedback from a discriminator network that the generator learns to create better images in an attempt to fool the discriminator into believing that the image from the generator is real. Now, in the DCGAN, the discriminator will be built using a convolutional neural network.

The following is an architecture diagram of our discriminator:




# Getting Ready

In this recipe, we will be heavily relying on the recipes in `Chapter 3`, *Convolutional Neural Networks for Computer Vision*, and so it would be best for you to quickly run through `Chapter 3`, *Convolutional Neural Networks for Computer Vision*.

# How to do it...

In this recipe, we will build the discriminator side of the GAN:

1. We will start with imports:

```
>>import torch.nn.functional as F
```

2. Then we will define the discriminator class:

```
>>class Discriminator_model(nn.Module):
    def __init__(self):
        super().__init__()
```

3. Next, we define discriminator units:

```
            self.disc = nn.Sequential(
                        nn.Conv2d(1, 32, 3, 2, 1),
                        nn.LeakyReLU(0.01),
                        nn.Conv2d(32, 64, 3, 2, 1),
                        nn.BatchNorm2d(64),
                        nn.LeakyReLU(0.01),
                        nn.Conv2d(64, 128, 3, 2, 1),
```

```
                nn.BatchNorm2d(128),
                nn.LeakyReLU(0.01)
            )
```

4. Then we define the last fully connected layer:

```
        self.fc = nn.Linear(2048, 1)
```

5. Then we define the `forward()` method:

```
    def forward(self, input):
        x = self.disc(input)
        return F.sigmoid(self.fc(x.view(-1, 2048)))
```

6. Then we create the discriminator object:

```
>>discriminator = Discriminator_model().to(device)
```

Now we have our discriminator ready.

# How it works...

In this recipe, we defined a classifier; used `nn.Sequential()` to define arrays of convolution, activation, and batch normalization units; and also defined a last fully connected layer that took in a flattened tensor and gave out a single output that went through a sigmoid layer. Since there are only two classes, we used the sigmoid layer in the end. The input is an image tensor of dimensions `1 x 28 x 28` and goes through the first convolution unit to give an output tensor of dimensions `32 x 14 x 14`. The second convolution layer makes it a `64 x 7 x 7` tensor, and then from there it becomes `128 x 4 x 4`; after that, we flatten and pass the tensor through the fully connected layer.

# See also

You can read about DCGAN at `https://arxiv.org/pdf/1511.06434.pdf`.

# Training a DCGAN model

We have defined the generator in our previous two recipes, *Creating a DCGan generator* and *Creating a DCGAN discriminator*. In this recipe, we will go on to train our GAN model. Remember that the goal of the generator is to create images that are as similar as possible to the dataset, and the goal of the discriminator is to distinguish between real and generated images. Theoretically, the generator captures all the features of the images in the dataset and cannot learn anything more, and the discriminator can only guess whether the image is real or generated. In this recipe, we will finish training our DCGANs model by integrating the generator and discriminator that we have created so far.

# Getting Ready

We will use the `torchsummary` library to see our model layers, their output shapes, and their parameters. For this, we will install the library using the following command:

```
pip install torchsummary
```

Once we are ready with this installation, we are good to move on to the recipe.

# How to do it...

In this recipe, we will complete our GAN training:

1. First, the imports:

```
>>from torchsummary import summary
>>import torch.optim as optim
>>import torchvision.utils as vutils
```

2. Then we initialize the weights of the generator and discriminator:

```
>>def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
>>generator.apply(weights_init)
>>discriminator.apply(weights_init)
```

3. Then we print the summary of our generator:

```
>>summary(generator, (100, ))
```

This gives us the following output:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                [-1, 12544]       1,266,944
   ConvTranspose2d-2          [-1, 128, 14, 14]         524,416
       BatchNorm2d-3          [-1, 128, 14, 14]             256
         LeakyReLU-4          [-1, 128, 14, 14]               0
   ConvTranspose2d-5           [-1, 64, 14, 14]          73,792
       BatchNorm2d-6           [-1, 64, 14, 14]             128
         LeakyReLU-7           [-1, 64, 14, 14]               0
   ConvTranspose2d-8            [-1, 1, 28, 28]           1,025
             Tanh-9            [-1, 1, 28, 28]               0
================================================================
Total params: 1,866,561
Trainable params: 1,866,561
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.97
Params size (MB): 7.12
Estimated Total Size (MB): 8.09
----------------------------------------------------------------
```

4. Then we will print the discriminator summary:

```
>>summary(discriminator, (1, 28, 28))
```

This gives us the following output:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 14, 14]             320
         LeakyReLU-2          [-1, 32, 14, 14]               0
            Conv2d-3            [-1, 64, 7, 7]          18,496
       BatchNorm2d-4            [-1, 64, 7, 7]             128
         LeakyReLU-5            [-1, 64, 7, 7]               0
            Conv2d-6           [-1, 128, 4, 4]          73,856
       BatchNorm2d-7           [-1, 128, 4, 4]             256
         LeakyReLU-8           [-1, 128, 4, 4]               0
            Linear-9                  [-1, 1]           2,049
================================================================
```

```
Total params: 95,105
Trainable params: 95,105
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.21
Params size (MB): 0.36
Estimated Total Size (MB): 0.58
----------------------------------------------------------------
```

5. Next, we will define our loss function:

```
>>criterion = nn.BCELoss()
```

6. We will also create a fixed noise:

```
>>fixed_noise = torch.randn(64, z_dim, device=device)
```

7. We will now define our optimizer functions:

```
>>doptimizer = optim.Adam(discriminator.parameters())
>>goptimizer = optim.Adam(generator.parameters())
```

8. We will then set our labels for the discriminator:

```
>>real_label, fake_label = 1, 0
```

9. We will also prepare to store the metrics from our training:

```
>>image_list = []
>>g_losses = []
>>d_losses = []
>>iterations = 0
>>num_epochs = 50
```

10. Now, we start our training loop:

```
>>for epoch in range(num_epochs):
```

11. Then we iterate through the data:

```
    print(f'Epoch : | {epoch+1:03} / {num_epochs:03} |')
    for i, data in enumerate(train_loader):
```

12. We then start training the discriminator by clearing its gradients:

```
        discriminator.zero_grad()
```

13. Then we fetch the images:

```
real_images = data[0].to(device)
size = real_images.size(0)
```

14. Then we create labels for these images:

```
label = torch.full((size,), real_label, device=device)
```

15. Next, we get the discriminator output:

```
d_output = discriminator(real_images).view(-1)
```

16. Then we calculate the discriminator error:

```
derror_real = criterion(d_output, label)
```

17. Next, we calculate the gradients:

```
derror_real.backward()
```

18. Now we will create a noise vector:

```
noise = torch.randn(size, z_dim, device=device)
```

19. Next, we pass the noise vector to the generator:

```
fake_images = generator(noise)
```

20. Then we create labels for the generated images:

```
label.fill_(0)
```

21. Then we pass them to the discriminator:

```
d_output = discriminator(fake_images.detach()).view(-1)
```

22. Next, we get the errors and gradients:

```
derror_fake = criterion(d_output, label)
derror_fake.backward()
derror_total = derror_real + derror_fake
```

23. We then update the discriminator weights:

```
doptimizer.step()
```

24. We start training the generator by clearing the gradients:

```
generator.zero_grad()
```

25. Then we change the labels from fake to real:

```
label.fill_(1)
```

26. Next, we get the discriminator output:

```
d_output = discriminator(fake_images).view(-1)
```

27. Then we calculate the generator loss and gradient and update the generator weights:

```
gerror = criterion(d_output, label)
gerror.backward()
goptimizer.step()
```

28. Then we save the losses:

```
if i % 50 == 0:
    print(f'| {i:03} / {len(train_loader):03} | G Loss:
{gerror.item():.3f} | D Loss: {derror_total.item():.3f} |')
    g_losses.append(gerror.item())
    d_losses.append(derror_total.item())
```

29. Then we save the images from the fixed noise:

```
if (iterations % 500 == 0) or ((epoch == num_epochs-1) and
(i == len(train_loader)-1)):
    with torch.no_grad():
        fake_images = generator(fixed_noise).detach().cpu()
        image_list.append(vutils.make_grid(fake_images,
padding=2, normalize=True))
iterations += 1
```

The following is a sample output:

```
Epoch : | 001 / 050 |
| 000 / 469 | G Loss: 1.939 | D Loss: 1.432 |
| 050 / 469 | G Loss: 3.920 | D Loss: 0.266 |
| 100 / 469 | G Loss: 3.900 | D Loss: 0.406 |
| 150 / 469 | G Loss: 3.260 | D Loss: 0.230 |
| 200 / 469 | G Loss: 3.856 | D Loss: 0.556 |
| 250 / 469 | G Loss: 4.097 | D Loss: 0.123 |
| 300 / 469 | G Loss: 2.377 | D Loss: 0.416 |
| 350 / 469 | G Loss: 2.984 | D Loss: 0.416 |
| 400 / 469 | G Loss: 3.262 | D Loss: 0.140 |
```

```
| 450 / 469 | G Loss: 3.469 | D Loss: 0.849 |
Epoch : | 002 / 050 |
| 000 / 469 | G Loss: 2.057 | D Loss: 0.484 |
| 050 / 469 | G Loss: 2.108 | D Loss: 0.435 |
| 100 / 469 | G Loss: 1.714 | D Loss: 0.862 |
| 150 / 469 | G Loss: 3.902 | D Loss: 0.199 |
| 200 / 469 | G Loss: 3.869 | D Loss: 0.086 |
| 250 / 469 | G Loss: 2.390 | D Loss: 0.208 |
| 300 / 469 | G Loss: 3.008 | D Loss: 0.586 |
| 350 / 469 | G Loss: 4.662 | D Loss: 0.074 |
| 400 / 469 | G Loss: 3.353 | D Loss: 0.368 |
| 450 / 469 | G Loss: 5.080 | D Loss: 0.110 |
Epoch : | 003 / 050 |
| 000 / 469 | G Loss: 7.159 | D Loss: 0.008 |
| 050 / 469 | G Loss: 5.087 | D Loss: 0.056 |
| 100 / 469 | G Loss: 4.232 | D Loss: 0.184 |
| 150 / 469 | G Loss: 5.037 | D Loss: 0.141 |
| 200 / 469 | G Loss: 5.636 | D Loss: 0.570 |
| 250 / 469 | G Loss: 3.624 | D Loss: 0.304 |
| 300 / 469 | G Loss: 4.291 | D Loss: 0.214 |
| 350 / 469 | G Loss: 2.901 | D Loss: 0.247 |
| 400 / 469 | G Loss: 3.703 | D Loss: 0.643 |
| 450 / 469 | G Loss: 1.149 | D Loss: 1.035 |
Epoch : | 004 / 050 |
| 000 / 469 | G Loss: 3.317 | D Loss: 0.202 |
| 050 / 469 | G Loss: 2.990 | D Loss: 0.350 |
| 100 / 469 | G Loss: 2.680 | D Loss: 0.162 |
| 150 / 469 | G Loss: 2.934 | D Loss: 0.391 |
| 200 / 469 | G Loss: 3.736 | D Loss: 0.215 |
| 250 / 469 | G Loss: 3.601 | D Loss: 0.199 |
| 300 / 469 | G Loss: 4.288 | D Loss: 0.164 |
| 350 / 469 | G Loss: 2.978 | D Loss: 0.086 |
| 400 / 469 | G Loss: 3.827 | D Loss: 0.189 |
| 450 / 469 | G Loss: 4.283 | D Loss: 0.216 |
Epoch : | 005 / 050 |
| 000 / 469 | G Loss: 4.456 | D Loss: 0.250 |
| 050 / 469 | G Loss: 4.886 | D Loss: 0.160 |
| 100 / 469 | G Loss: 1.844 | D Loss: 0.447 |
| 150 / 469 | G Loss: 3.680 | D Loss: 0.505 |
| 200 / 469 | G Loss: 4.428 | D Loss: 0.200 |
| 250 / 469 | G Loss: 4.270 | D Loss: 0.222 |
| 300 / 469 | G Loss: 4.617 | D Loss: 0.102 |
| 350 / 469 | G Loss: 3.920 | D Loss: 0.092 |
| 400 / 469 | G Loss: 4.010 | D Loss: 0.392 |
| 450 / 469 | G Loss: 1.705 | D Loss: 0.651 |
```

With this, we have finished training our DCGAN.

# How it works...

We start with the `weights_init` function, which is used to initialize all the weight randomly from a normal distribution with a mean of 0 and a standard deviation of 0.02. After the model is initialized the function takes the model as input and reinitializes all convolutional, convolutional-transpose, and batch-normalization layers.

We then printed the summary of our model using the `torchsummary` library; to do this, we passed the model along with the input dimension. This is handy to see whether all our output dimensions are correct and to check the number and size of parameters in each layer. Next up, we defined the loss function, and we used binary cross entropy loss, since there is only one output with two possible states, denoting whether the image is real, 1, and fake, 0.

We also created a fixed noise, which we used to visualize the GAN model improvement over the iterations. We used the ADAM optimizer to update the weights of both the generator and discriminator, `goptimizer` and `doptimizer`, respectively. We then made provisions to store some of the model metrics to see how the model changed over the iterations, and then we started the training loop.

We iterated through each of the mini batches and started training the discriminator. We took only the image from the MNIST dataset and moved it to the device with `real_images = data[0].to(device)`; since the images are all from the MNIST dataset, we knew that they are real, and so we created a label vector of the same size as the mini batch and filled it with the real image label, 1. We then passed these real images into the discriminator for prediction, and then used this prediction to get the error, `derror_real`, from the criterion and calculate the gradient. We then created an equal number of noise vectors and passed them to the generator to produce images, and then we passed these generated images into the discriminator to get the predictions and then the error from the criterion, `derror_fake`. Then we did this again to calculate and accumulate the gradients. We then got the sum of the error from the real and fake images to get the total discriminator error and also update the weights of the discriminator.

We then started training the generator, and the generator should have been able to fool the discriminator. The generator has to correct cases where the discriminator rightly predicted generated images as fake. Therefore, whenever the prediction from the discriminator labels a generated image as fake, that adds to the generator loss, `gerror`. We then calculated the gradients and updated the generator weights.

We then displayed the model metrics at regular intervals and also saved the images generated by the generator of the fixed noise to visualize the model's performance over the epochs.

## There's more...

You can play around more with the hyperparameters of the network—for instance, you could use different learning rates for the discriminator optimizer than those of the generator or train the generator twice or three times for every update of the discriminator.

## See also

You can see a different example of training and architecturing DCGAN at `https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html` and `https://iq.opengenus.org/deep-convolutional-gans-pytorch/`.

# Visualizing DCGAN results

We have previously seen that, with a GAN, the generator and discriminator are competing against each other, and by doing this, they create better and better images; however, theoretically, they reach a point where the generator has captured all the features of the real image and there is nothing more that a generator can learn; similarly, the discriminator can only guess whether the given image is real or fake with a 50/50 chance of success. At this point, the GAN is said to have converged.

Now, any improvement on one side leads to degradation in the result of the other side, which is a **zero-sum** state or a **Nash equilibrium**; however, in practice this is hard to achieve, as both the generator and discriminator are continuously changing, and so the best way to check the GAN performance is through plots and graphs. In this recipe, we will have a quick look at visualization.

## Getting Ready

For this recipe, you must have the `matplotlib` and `numpy` libraries, which you can install using `pip` as follows:

```
pip install matplotlib
pip install numpy
```

With these installed, we will move on to the recipe.

# How to do it...

In this recipe, we will quickly plot the graphs and images from the GAN:

1. We will start with imports:

```
>>import matplotlib.pyplot as plt
>>import numpy as np
```

2. Then we will add the plot size and title:

```
>>plt.figure(figsize=(10,5))
>>plt.title("Generator and Discriminator Loss During Training")
```

3. Next, we add the generator and discriminator losses:

```
>>plt.plot(g_losses,label="Generator")
>>plt.plot(d_losses,label="Discriminator")
```

4. Then we add the *x* and *y* axis labels:

```
>>plt.xlabel("iterations")
>>plt.ylabel("Loss")
```

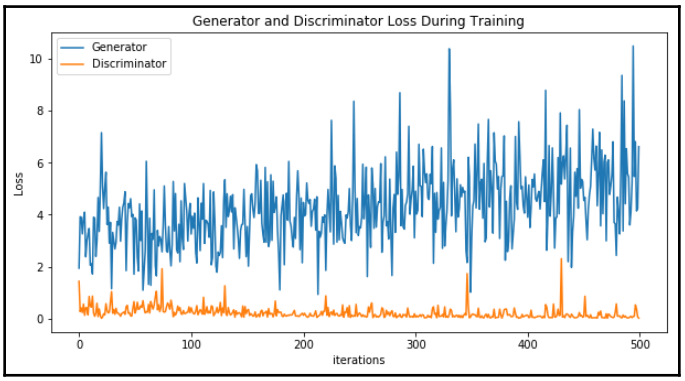
5. Next, we add a legend for the graph:

```
>>plt.legend()
```

6. Finally, we show the plot:

```
>>plt.show()
```

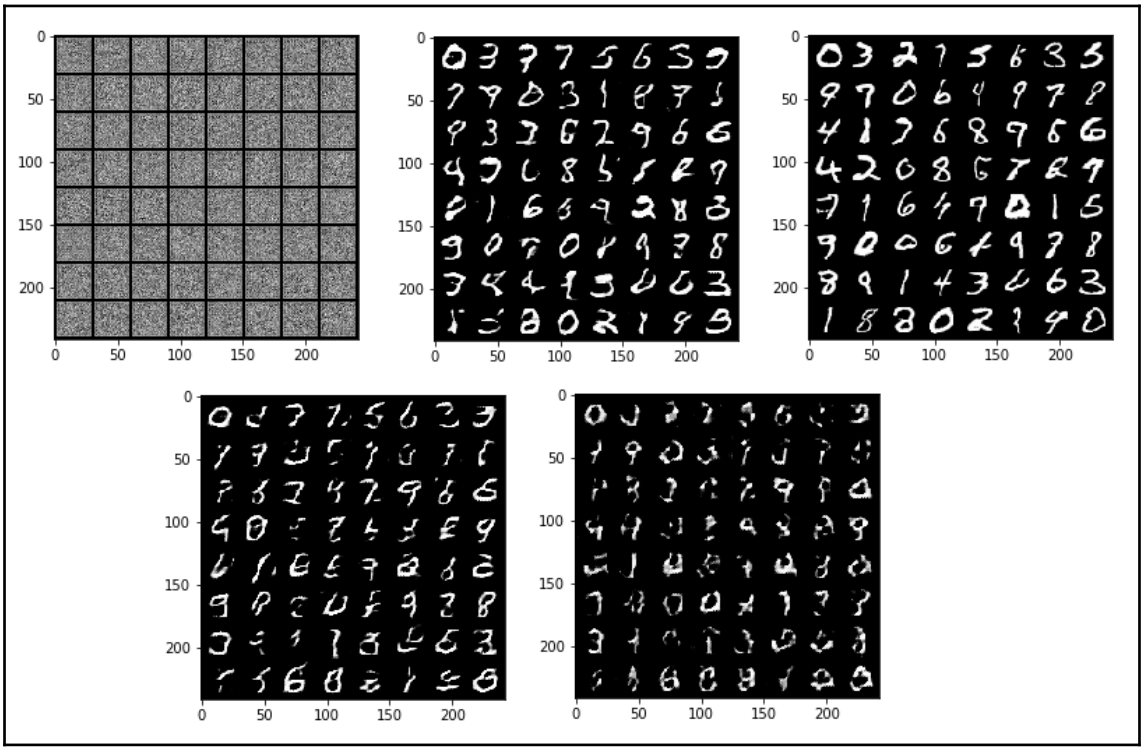
This will give the following output:



7. Then, we iterate through the images in the `image_list` and show them:

```
>>for image in image_list:
    plt.imshow(np.transpose(image,(1,2,0)))
    plt.show()
```

This results in the following output:



Here, we saw the hand-written image generated by our DCGAN.

# How it works...

In this recipe, we used matplotlib to plot the graph and image. We set the figure dimensions and title using the `figure()` and `title()` methods, and then plotted the generator and discriminator losses using the `plot()` method. We also added the *x* and *y* labels using the `xlabel` and `ylabel` methods. We also added a legend for the graph using the `legend()` method and finally showed the plot using the `show()` method.

We iterated through the images we saved in `image_list` during the training, and we used NumPy's `transpose()` method to fix the dimensions of the images in the desired order. The images in `image_list` were generated using the `torchvision.util.make_grid()` method, and we created a grid of generated images from the noise vector.

## There's more...

You could use other libraries, such as plotly and seaborn, to plot and beautify graphs.

## See also

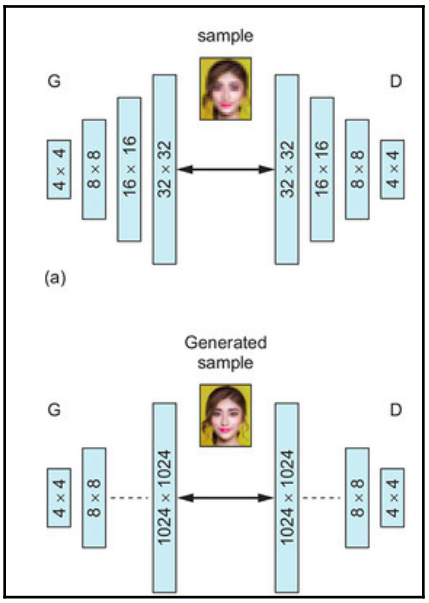
You can see a visualization and animation for DCGAN at `https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html#results`.

# Running PGGAN with PyTorch hub

In this recipe, we will look at **progressive GANs** (**PGGANs**), which are advanced GANs compared to DCGANs, and are capable of generating photorealistic images. A PGGAN trains the GAN network in multiple phases. It takes in a latent feature, *z*, and uses two deconvolution layers to generate 4 × 4 images. On the discriminator side, the network trains with the generated 4 x 4 images using two convolution layers. After the network is stable, it adds two more convolution layers to upsample the images to 8 x 8, and two more convolution layers to downsample images in the discriminator.

After nine such progressions, we will have 1024 x 1024 images generated by the generator. The progressive training strategy of PGGAN has an advantage over a regular GAN, as it speeds up and stabilizes the training. The speed is due to the fact that most of the training happens at a lower resolution, and the progression to higher resolution happens after the network achieves stability in each phase.

The following is an abstract representation of PGGAN:




The key innovations of PGGAN can be summarized as follows:

- **Progressively growing and smoothly fading in higher-resolution layers**: It goes from low-resolution to high-resolution convolutions, and rather than immediately jumping the resolution, it smoothly fades a nedonew layer with a higher resolution by a parameter of alpha (α) (which is between 0 and 1) that controls how much we use either the old or the upscaled larger output.
- **Mini batch standard deviation**: We calculate a statistic for the discriminator, which is the standard deviation of all the pixels in the mini batch that are generated by the generator or that come from the real data. Now the discriminator needs to learn that, if the standard deviation is low in images from the batch it is evaluating, the image is likely fake, because the real data will have a higher variance. Therefore, the generator has to increase the variance of the generated samples to fool the discriminator.
- **Equalized learning rate**: All the weights ($w$) are normalized ($w'$) to be within a certain range so that $w' = w/c$ by a constant $c$ that is different for each layer, depending on the shape of the weight matrix.
- **Pixelwise feature normalization**: The feature vector in each pixel is normalized, since the batch norm works best with large mini batches and is memory-intensive.

The original implementation of PGGAN by Nvidia took one to two months to run; however, in order for us to see the performance of PGGAN, we will use the PyTorch hub, which is a repository of a pretrained model built using PyTorch.

# Getting ready

For the torch hub to work, you will need to have PyTorch version 1.1.0 or above.

# How to do it...

In this recipe, we will run PGGAN from the torch hub:

1. First, we will set up imports:

```
>>import torch
>>import matplotlib.pyplot as plt
>>import torchvision
```

2. Then we check the GPU:

```
>>use_gpu = True if torch.cuda.is_available() else False
```

3. Next, we load the pretrained PGGAN model:

```
>>model = torch.hub.load('facebookresearch/pytorch_GAN_zoo:hub',
'PGAN',
                        model_name='celebAHQ-512',
                        pretrained=True,
                        useGPU=use_gpu)
```

4. Then we generate noise:

```
>>num_images = 5
>>noise, _ = model.buildNoiseData(num_images)
```

5. Next, we get the generator output

```
>>with torch.no_grad():
    generated_images = model.test(noise)
```

6. Finally, we make the image grid and display it:

```
>>grid = torchvision.utils.make_grid(generated_images.clamp(min=-1,
max=1), scale_each=True, normalize=True)
>>plt.imshow(grid.permute(1, 2, 0).cpu().numpy())
```

Here, we see the generated face images from PGGAN:


With the previous steps, we have learned how to run PGGAN using PyTorch hub.

# How it works...

In this recipe, we loaded the PGGAN pretrained model that was trained on the `celebAHQ` dataset; we used the `load()` method in `torch.hub` for this. We then defined the number of images we need to create and generate a noise vector, which had the dimension of `num_images x 512`, since this model is trained with a noise vector of size 512, which is all handled internally by the `buildNoiseData()` method available in the model object.

The `model.test()` method generated the images with which we make a grid. The clamp method limited all the values in the range defined by `min` and `max`. The `.cpu()` method moved the generated image to the CPU, and we used `permute` to fix the dimensions. Finally, `plt.imshow()` displayed the grid we created.

# There's more...

You can explore the complete PGGAN implementation at `https://github.com/github-pengge/PyTorch-progressive_growing_of_gans`.

# See also

You can learn more about the torch hub at `https://pytorch.org/docs/stable/hub.html`.

You can see the PGGAN implementation from Nvidia at `https://github.com/tkarras/progressive_growing_of_gans`, `https://research.nvidia.com/publication/2017-10_Progressive-Growing-of`.

# 7
# Deep Reinforcement Learning

In this chapter, we will explore the application of **neural networks** (**NNs**) on **reinforcement learning** (**RL**) using PyTorch.

RL is a domain of **artificial intelligence** (**AI**) that is different from the other machine learning formats that we looked at in the previous chapters. RL is a subclass of machine learning algorithms that learns by maximizing the rewards in an environment. These algorithms are useful when the problem involves making decisions or taking actions.

In this chapter, we will cover the following recipes:

- Introducing OpenAI gym – CartPole
- Introducing DQNs
- Implementing the DQN class
- Training a DQN
- Introducing Deep GAs
- Generating agents
- Selecting agents
- Mutating agents
- Training a Deep GA

# Introducing deep RL

The **agent** is the core of any RL problem. It is the part of the RL algorithm that processes input information in order to perform an action. It explores and exploits knowledge from repeated trials in order to learn how to maximize the reward. The scenario that the agent has to deal with is known as the **environment**, while **actions** are the possible moves that an agent can make in a given environment. The return from an environment upon taking an action is called the **reward**, and the course of action that the agent applies to determine the next action based on the current state is called the **policy**. The expected long-term return with a discount, as opposed to the short-term reward, is called the **value**. The **q-value** is similar to the value but has an additional current action parameter.

Now that we have looked at the context of RL, let's understand why we should use **deep RL** (**DRL**) as opposed to RL. DRL combines RL with deep learning, which uses NNs to solve RL problems.

A deep learning algorithm can learn to abstract away the details of the states of an environment and then learn the important features of a state. Since a deep learning algorithm only has a finite number of parameters, we can use it to compress possible states into fewer states, and then use that new representation to pick an action.

An RL solution involves storing the results from trial and error in a lookup table, which becomes huge when the environment becomes more and more complex. A deep neural network might learn to recognize the same high-level features a programmer would have to hand-engineer in a lookup table approach on its own.

Deep learning is behind the recent breakthroughs in RL. They exhibit representational power, efficiency, flexibility, and lead to a simplified solution for the problem at hand.

# Introducing OpenAI gym – CartPole

In this recipe, we will be implementing two different RL algorithms. We will need an environment to run our algorithms in so that the model we create will have a maximum reward.

We will be using OpenAI's gym library, which is a collection of environments that we can use to train our model. We will be focusing on a specific environment called `Cartpole-v1`.

A cartpole is an inverted pendulum with a center of gravity above its pivot point. We have to move the pivot point under the center of mass in order to control this unstable position of the inverted pendulum. The goal is to apply appropriate forces in order to keep the cartpole balanced on the pivot point.

The following diagram shows OpenAI gym's Cartpole:

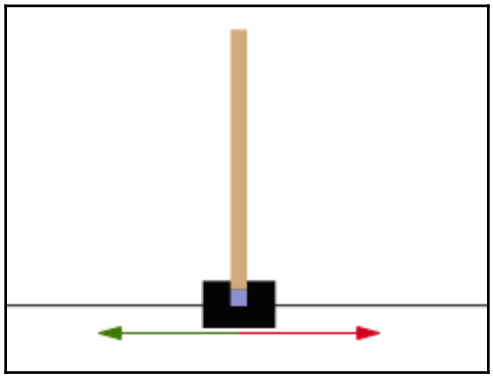


Figure 1: OpenAI gym – Cartpole

The cartpole is attached to the cart with an un-actuated joint that moves freely on a horizontal track. The pendulum starts by standing perpendicular to the horizontal track. The goal is to prevent it from falling over by applying the forces +1 and -1.

The cartpole is considered to have failed to acheive the goal when the pole is more than 15 degrees from the vertical position or the cart moves more than 2.4 units from the center.

For every timestep that the pole remains upright, a reward of +1 is achieved. Now that we have a context, we will try out the code for OpenAI gym's Cartpole problem.

# Getting ready

First, we need to install `gym`. We will install it using the `pip` manager:

```
pip install gym
```

Now that we have installed `gym`, let's jump into using the `gym` library.

# How to do it...

In this recipe, we will understand the cartpole environment. Follow these steps to do so:

1. We will start by importing the `gym` module:

```
>>import gym
```

2. Next, we will make the environment:

```
>>env = gym.make('CartPole-v0')
```

3. Next, we will reset the environment:

```
>>env.reset()
```

4. Now, we will render the environment:

```
>>for _ in range(1000):
    env.render()
    action = env.action_space.sample()
    observation, reward, done, info = env.step(action)
    if done:
        env.reset()
>>env.close()
```

This will give us the following output:

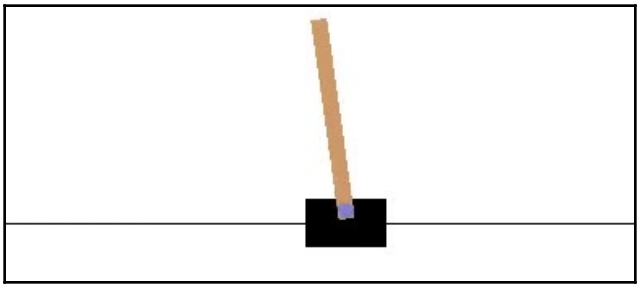


Figure 2: Output of the cartpole environment

5. Get the number of possible actions that can be performed:

```
>>env.action_space.n
2
```

6. Get the number of observation state parameters:

```
>>env.observation_space.shape[0]
4
```

By doing this, you will see a window that shows a cartpole that is unstable and where the pivot point moves randomly.

# How it works...

In this recipe, we explored some of the functionalities of OpenAI Gym. We started by making an environment using `Cartpole-v0`, which has a maximum achievable score of 200, after which the environment terminates. We brought the environment to an initial state where the cartpole is in the upright position using the `env.reset()` command. Then, we started a loop for 1,000 steps, wherein we rendered the current environment's start using `render()` and we chose a random action for the current state using `env.action_space.sample()`. Then, we passed the selected action into the `step` method of the environment. The `step` method tells us what happened to the environment when we performed the current action on the current state of the environment. The `step` method returns the following:

- **Observation**: This is an object that tells us about the new state of the environment. The object is specific to the environment we selected.
- **Reward**: This gives us the reward that was achieved by the selected action. In the case of the cartpole, this is 1.0 for every timestep when the cartpole is upright and 0.0 otherwise.
- **Done**: This is a Boolean value that tells us whether the environment has reached the terminal state, either from failing the task at hand, which in the case of the cartpole is when the pole fails to hold the upright position, or from completing the task at hand, which in the case of the cartpole is when it reaches the max timestep of 200.
- **Info**: Diagnostic information that's useful for debugging.

In our loop, whenever the pole tips over, we reset the environment to its initial state.

Finally, we closed the environment. We looked at the number of possible actions in an environment using `env.action_space.n` and the number of parameters in the observation state using `env.observation_space.shape[0]`. Now that we have an understanding of the environment, we can start implementing various deep RL algorithms.

# There's more...

You can try other environments by changing the environment name. It would be beneficial for you to try out `Cartpole-v1`.

# See also

You can read more about OpenAI gym at `http://gym.openai.com/docs/`.

# Introducing DQNs

Before we jump into the next recipe, let's have a quick look at DQNs.

A DQN is an RL technique that aims at picking the best possible action for a given observation. There is a q-value, which is the quality of a given move that's associated with each possible action for each possible observation. In the traditional RL algorithm, this q-value comes from a q-table, which is a lookup table, where it is a table holding q-values. This lookup table is updated iteratively by playing the game over and over and using the reward to update the table. The q-learning algorithm learns the optimum values to be populated in this table. We can simply look at the table for a given state and select the action with the maximum q-value in order to maximize the chance of winning the game.

The q-value can be updated as follows:

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left( \underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

The new q-value is a sum of two parts. The first part is **(1-learning rate)*old Q value**, which is how much of the old value will be remembered. With a 0 learning rate, nothing new will be learned and with a learning rate of 1, all of the old values will be forgotten.

The second part is **learning rate * (immediate reward for action + discounted estimate of optimal future value)**, where the learned value is the immediate reward plus the discounted estimate of the optimal future value. The importance of future rewards is decided by the discount factor.

With Deep Q-learning, instead of using a Q table to look up the action with a maximum possible q-value for a given state, we use a deep neural network to predict the Q-values for the actions and pick the action with the maximum q-value for a given action.

# How to do it...

In this recipe, we will define our neural network model for the cartpole problem. Follow these steps to do so:

1. First, we will import `torch`:

```
>>import torch
>>import torch.nn as nn
```

2. Next, define a function to return the model:

```
def cartpole_model(observation_space, action_space):
    return nn.Sequential(
        nn.Linear(observation_space, 24),
        nn.ReLU(),
        nn.Linear(24, 24),
        nn.ReLU(),
        nn.Linear(24, action_space)
    )
```

This function returns the cartpole model.

# How it works...

In this recipe, we defined a function called `cartpole_model`, which takes in the `observation_ space` and `action_space` parameters and returns a neural network model. Here, we used the `Sequential` module from `torch.nn` and `nn.Linear` and `nn.ReLU` to complete the model. We used this model to train and predict the q-values for each action, given an observation.

# There's more...

We could also train a model that takes in the state as an image and learns to predict q-values from the image. After doing this, we would use `nn.Conv2d()` in order to use convolutional neural networks.

## See also

You could look at an alternative architecture at `https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html#q-network`.

# Implementing the DQN class

In this recipe, we will complete the DQN using our neural network. To do so, we will perform some key tasks, including creating the target and policy networks, the loss function, and the optimizers for the network, storing the states and rewards of the learning process, predicting the action, experience replay, and controlling the exploration rate.

## Getting ready

Before you complete this recipe, you should complete the *Introducing OpenAI gym – Cartpole* recipe of this chapter in order to set up the `gym` package.

## How to do it...

In this recipe, we will look at all the key functionalities we can use to perform DQN. Follow these steps:

1. We will start with the necessary imports:

```
>>import random
>>from collections import deque
>>import numpy as np
>>import torch.optim as optim
```

2. Next, we will define our DQN class:

```
>>class DQN:
```

3. Then, we will define the constructor:

```
    >>def __init__(self, observation_space, action_space):
            self.exploration_rate = MAX_EXPLORE
            self.action_space = action_space
            self.observation_space = observation_space
            self.memory = deque(maxlen=MEMORY_LEN)
```

4. Next, we will define `target_net` and `policy_net`:

```
self.target_net =
cartpole_model(self.observation_space,
self.action_space)
self.policy_net =
cartpole_model(self.observation_space,
self.action_space)
```

5. Now, we will copy the weights:

```
self.target_net.load_state_dict(self.policy_net.state_
dict())
self.target_net.eval()
```

6. Here, we define the loss function, optimizer, and limit flag:

```
self.criterion = nn.MSELoss()
self.optimizer =
optim.Adam(self.policy_net.parameters())

self.explore_limit = False
```

7. Next, we will define the `load_memory` method:

```
>>def load_memory(self, state, action, reward, next_state,
terminal):
        self.memory.append((state, action, reward, next_state,
terminal))
```

8. Now, we will define the `predict_action` method:

```
>>def predict_action(self, state):
        random_number = np.random.rand()
        if random_number < self.exploration_rate:
            return random.randrange(self.action_space)
        q_values = self.target_net(state).detach().numpy()
        return np.argmax(q_values[0])
```

9. Now, we will jump to the `experience_replay` method:

```
>>def experience_replay(self):
        if len(self.memory) < BATCH_SIZE:
            return

        batch = random.sample(self.memory, BATCH_SIZE)
```

10. Now, let's update the q-value using the batch:

```
for state, action, reward, next_state, terminal in
batch:
    q_update = reward
    if not terminal:
        q_update = reward + GAMMA *
self.target_net(next_state).max(axis=1)[0]
    q_values = self.target_net(state)
    q_values[0][action] = q_update
```

11. Next, we calculate the loss and update the weights:

```
    loss = self.criterion(self.policy_net(state),
    q_values)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

12. We will also update the exploration rate:

```
    if not self.explore_limit:
        self.exploration_rate *= EXPLORE_DECAY
        if self.exploration_rate < MIN_EXPLORE:
            self.exploration_rate = MIN_EXPLORE
            self.explore_limit = True
```

With this, we have completed the DQN class.

# How it works...

In this recipe, we completed the DQN class and added all the required functionality to train the DQN. In the constructor, we initialized the initial state of exploration, the observation space, and the action space, and then we defined a memory unit to hold the experiences of DQN. We created two instances of the cartpole model called `policy_net` and `target_net`. We need two networks since, at every step of training, the Q-network's values shift, and the network can become destabilized by falling into feedback loops between this changing target and the estimated Q-values if we use constantly shifting target values to adjust our network values. If this happens, the value estimations will spiral out of control. Due to this, we used two networks and kept `target_net` in `eval` mode. Then, we used `MSELoss()` as the loss function along with the `Adam` optimizer to update the weights.

In the `load_memory()` method, we stored the state, action, reward, next state, and terminal from the environment to be used to train the network. The next method we used was `predict_action`. In this method, we picked a `random_number` using `np.random.rand()`, which gives us a value from `[0,1)`. If this `random_number` is less than the current `exploration_rate`, then we pick a random action, which is controlled by `exploration_rate`. This is how we incorporate exploration. However, if `random_number` is greater than the `exploration_rate`, then the `target_net` predicts the `q_values` and picks the action with the maximum q-value.

Finally, we implemented the `experience_replay` method. Here, we wait for the datapoints to be at least `BATCH_SIZE` in number and then randomly sampled a batch from the memory. This allows it to learn from a varied set of observations, rather than from a sequence of closely related observations. As we iterate through the batch, we updated the q-values based on the q-value updation formula using our `target_net`. Then, we trained the `policy_net` based on the error between the new q-value and the one predicted by the `policy_net`. After that, we reduced the exploration rate gradually by multiplying it with exploration decay until we got the minimum exploration rate. We did this because, at the initial phase of training, we want the agent to explore more; however, as we progress with training, we want the algorithm to converge. With this, we've completed all the functionalities of the DQN class.

## There's more...

You can add a method within the DQN class to update the weights of the target network with the policy network.

## See also

You can view an implementation of this in Keras at `https://github.com/gsurma/cartpole/blob/master/cartpole.py`.

# Training DQN

In this recipe, we will finish training our DQN reinforcements algorithm and visualize our cartpole after training the model. We will use our DQN class to predict the action and apply that action to the environment to obtain the reward. The goal of this recipe is to maximize the reward. We will use experience replay to train our model to predict q-values.

# How to do it...

In this recipe, we will continue with the *DQN class* recipe. Follow these steps:

1. First, we need to import `gym`:

```
>>import gym
```

2. Next, we need to initialize the constants and the environment:

```
>>ENV_NAME = "CartPole-v1"
>>BATCH_SIZE = 20
>>GAMMA = 0.95
>>LEARNING_RATE = 0.001
>>MAX_EXPLORE = 1.0
>>MIN_EXPLORE = 0.01
>>EXPLORE_DECAY = 0.995
>>MEMORY_LEN = 1_000_000
>>UPDATE_FREQ = 10
```

3. Now, we need to initialize the environment and DQN:

```
>>env = gym.make(ENV_NAME)
>>observation_space = env.observation_space.shape[0]
>>action_space = env.action_space.n
>>dqn = DQN(observation_space, action_space)
```

4. Now, let's start the training loop:

```
>>for i in range(100):
    state = env.reset()
    state = np.reshape(state, [1, observation_space])
    state = torch.from_numpy(state).float()
```

5. Next, we need to step through the environment:

```
score = 0
while True:
    score += 1
    action = dqn.predict_action(state)
    next_state, reward, terminal, info = env.step(action)
```

6. Here, we have to find the next state:

```
next_state =
torch.from_numpy(np.reshape(next_state, [1,
observation_space])).float()
dqn.load_memory(state, action, reward, next_state,
```

```
terminal)
state = next_state
```

7. We will use the following code to end the infinite loop:

```
if terminal:
    print(f'| {i+1:02} |
{dqn.exploration_rate:.4f} | {score:03} |')
    break
```

8. Next, we need to perform an experience replay:

```
dqn.experience_replay()
```

9. Next, update the weights:

```
if steps%UPDATE_FREQ == 0:
dqn.target_net.load_state_dict(dqn.policy_net.stat
e_dict())
```

The following code block shows some sample outputs:

```
| Run | Exploration Rate | Score |
| 001 |       0.9416     | 032   |
| 002 |       0.8956     | 011   |
| 003 |       0.8061     | 022   |
| 004 |       0.7477     | 016   |
| 005 |       0.6936     | 016   |
| 006 |       0.6498     | 014   |
| 007 |       0.5371     | 039   |
.
.
| 072 |       0.0100     | 256   |
| 073 |       0.0100     | 227   |
| 074 |       0.0100     | 225   |
| 075 |       0.0100     | 238   |
| 076 |       0.0100     | 154   |
| 077 |       0.0100     | 285   |
.
.
.
```

10. Now, we will define a function that will visualize the cartpole's performance:

```
>>def play_agent(dqn, env):
    observation = env.reset()
    total_reward=0
```

We need to use the following code to iterate for a maximum of 500 steps:

```
        for _ in range(500):
            env.render()
            observation =
    torch.tensor(observation).type('torch.FloatTensor').view(1,-1)
            q_values = dqn.target_net(observation).detach().numpy()
            action = np.argmax(q_values[0])
            new_observation, reward, done, _ = env.step(action)
            total_reward += reward
            observation = new_observation

            if(done):
                break
```

Finally, close the environment:

```
        env.close()
        print("Rewards: ",total_reward)
```

Invoke the `play_agent()` function:

```
    >>play_agent(dqn, env)
    Rewards: 160.0
```

With this, we've trained and visualized our DQN.

# How it works...

In this recipe, we started by importing and initializing our environment with hyperparameters. Then, we created an instance of the `DQN` class and started the training loop, reset the environment, and reshaped the state array so that it can be fed into the model as a float tensor. Then, we started an infinite loop that terminates when the return from the `env.step()` method has `terminal` set to `True`. The `predict_action()` method predicts the action to be taken given the current state of the environment. Then, we applied this action using the `step()` method of the environment. We took the next state that was returned by the step method and converted it from a `numpy` into a `torch.FloatTensor` and saved the parameters of the environment. We passed this new state into the model over and over again. We also copied the weights from our policy net to the target net every few steps.

Finally, we wrote a simple `play_agent` function to visualize the cartpole balancing and ran a loop after resetting the environment. We asked the target net to predict the q-values for each of the possible actions, picked the action that has the highest q-value, and used `step()` to step that action into the environment. After that, we kept adding the reward. This function returns the number of timesteps the cartpole was upright and a video of the cartpole performing the balancing act.

## There's more...

You can write a function to plot the algorithm's performance and only stop the training process when the score of the model consistently hits between 450-500.

## See also

You see an implementation of this in Keras at `https://github.com/gsurma/cartpole/blob/master/cartpole.py`.

You can see an alternative implementation at `https://github.com/pytorch/tutorials/blob/master/intermediate_source/reinforcement_q_learning.py`.

# Introduction to Deep GA

In the recipe, we will explore the **deep genetic algorithm** (**Deep GA**) and show you that it is a competitive alternative to gradient-based methods when applied to RL. Instead of having a randomly generated network whose weights are modified using gradient descent, we will generate a set of networks randomly, creating a generation that is then evaluated on its performance in a given environment. Note that some networks in a generation will perform slightly better than others. We will pick the networks that performs the best and keep it for the next generation of networks. Then, we'll create the next generation by copying them and make random modifications to their weights. Since we'll be picking the top-performing networks with minor changes to the weights the network's overall performance will keep getting better.

In this recipe, we will define the network dimensions of the model that will be used to make predictions on what action should be taken when given a state.

# How to do it...

In this recipe, we will complete the network model definition. You need to have OpenAI's gym library installed. Follow these steps:

1. We will start with the imports:

```
>>import torch.nn as nn
>>import torch
```

2. Now, let's define a function that returns a neural network:

```
>>def cartpole_model(observation_space, action_space):
    return nn.Sequential(
        nn.Linear(observation_space, 128),
        nn.ReLU(),
        nn.Linear(128, action_space),
        nn.Softmax(dim=1)
    )
```

With this, we've completed the model's definition.

# How it works...

In this recipe, the function took in an observation state and ran it through two linear layers and a ReLU unit, with 128 units in the hidden layer. The output of the final layer was passed through a softmax function to convert the activation into probabilities and to choose the action with the highest probability.

# There's more...

You could also have multiple layers and different number of units for complex models.

# See also

You can also use a complex network with a convolutional layer. An example of this is shown at `https://github.com/IBM/distributed-evolutionary-ml/blob/master/nn.py`.

# Generating agents

In this recipe, we will look at creating a set of agents to start our evolution process and then initializing the weights of these agents. We will use these agents to evaluate the performance of the model and to generate the next generation of agents.

# How to do it...

In this recipe, we will create a given number of agents. Follow these steps:

1. First, we will define a function that will initialize the weights of the agents:

```
>>def init_weight(module):
    if((type(module) == nn.Linear)):
            nn.init.xavier_uniform_(module.weight.data)
            module.bias.data.fill_(0.00)
```

2. Now, we will define a function that will create the agents:

```
>>def create_agents(num_agents, observation_space, action_space):
      agents = []
```

3. Next, we will create `num_agents` number of agents:

```
          for _ in range(num_agents):
              agent = cartpole_model(observation_space, action_space)
              agent.apply(init_weight)
```

4. We will turn off the gradients for each of the layers of the agents:

```
              for param in agent.parameters():
                  param.requires_grad = False

              agent.eval()
              agents.append(agent)
```

5. Finally, we'll return the agents:

```
          return agents
```

Now, our agents are ready to be evaluated.

## How it works...

In this recipe, we wrote two functions—the first one initializes the weights of the layer of the model. For the model weights, we used `xavier_uniform` from `torch.nn.init` and filled the bias with `0`. The second function creates `num_agents` number of agents and returns them using the `cartpole_model()` function. We initialized the weights using `init_weight`. Then, for the parameters of the model, we disabled the gradient calculation, set the agents in `eval()` mode, and returned all the agents.

## See also

You can find out about other initialization methods at `https://pytorch.org/docs/stable/nn.init.html`.

# Selecting agents

In this recipe, we will look at agent selection based on the fitness function, which in our case means having a high score for balancing our cartpole. This means we'll propagate the agents that have the top scores and ignore the rest. We will evaluate each agent in a given generation and evaluate them more than once to ensure that the reward was not by chance. Finally, we will use the average of the scores from each of agents to identify the best-performing agents.

## How to do it...

In this recipe, we will write the functions for evaluating an agent, evaluating it multiple times, and evaluating all of the agents in a given sequence. Follow these steps:

1. We will start with the imports:

```
>>import numpy as np
```

2. Next, we need to define a function that will evaluate the agent's performance:

```
>>def eval_agent(agent, env):
    observation = env.reset()
```

3. Next, we need to run the loop for the maximum possible timestep:

```
total_reward = 0
for _ in range(MAX_STEP):
    observation =
torch.tensor(observation).type('torch.FloatTensor').view(1,-1)
    action_probablity = agent(observation).detach().numpy()[0]
    action = np.random.choice(range(env.action_space.n), 1,
p=action_probablity).item()
    next_observation, reward, terminal, _ = env.step(action)
    total_reward += reward
    observation = next_observation
    if terminal:
        break
return total_reward
```

4. Then, we need to define the average agent score:

```
>>def agent_score(agent, env, runs):
    score = 0
    for _ in range(runs):
        score += eval_agent(agent, env)
    return score/runs
```

5. Finally, we evaluate the scores of all the agents:

```
>>def all_agent_score(agents, env, runs):
    agents_score = []
    for agent in agents:
        agents_score.append(agent_score(agent, env, runs))
    return agents_score
```

Now, our functions are ready to be evaluated.

# How it works...

In this recipe, we completed some of the key functionalities of the deep genetic algorithm. We looked at three different functions—the first function, `eval_agent()`, is very similar to the what we saw in the *Training DQN* recipe, wherein we used the agent, which is a neural network model, that predicts the action to take and runs until `MAX_STEP` (for `cartpole-v1`, this is 500) or the Terminal is `True` and returns the score.

Then, we used the second function, `agent_score()`, to return an average score over the specified number of `runs` and returned this average score to ensure the model didn't perform well at random. The last function, `all_agent_score()`, simply loops through all of the agents in a generation and gets the average score for all of the agents in a generation.

# Mutating agents

In this recipe, we will look at mutating agents. After we've picked the best-performing model from a given generation and before creating the next generation of agents, we will introduce slight random variation to the weights of these selected agents, which allows the agent to explore more regions for better rewards, just like how biological evolution works.

# How to do it...

In this recipe, we will identify elite agents and add mutations to these agents. Follow these steps:

1. First, we will import the `copy` and `numpy` modules:

```
>>import copy
>>import numpy
```

2. Next, we will define the `mutation` function:

```
>>def mutation(agent):
    child_agent = copy.deepcopy(agent)
```

3. Next, we'll iterate through the parameters of the agents:

```
    for param in agent.parameters():
        mutation_noise = torch.randn_like(param) * MUTATION_POWER
```

4. Then, we add the mutation noise to the parameters:

```
        param += mutation_noise
    return child_agent
```

5. Now, define the `elite` function:

```
>>def elite(agents, top_parents_id, env, elite_id=None, top=10):
    selected_elites = top_parents_id[:top]
    if elite_id:
        selected_elites.append(elite_id)
```

```
    top_score = np.NINF
    top_id = None
```

6. Next, find the `elite` agent:

```
    for agent_id in selected_elites:
        score = agent_score(agents[agent_id], env, runs=5)
        if score > top_score:
           top_score = score
           top_id = agent_id
    return copy.deepcopy(agents[top_id])
```

7. Get the child agents:

```
>>def child_agents(agents, top_parents_id, env, elite_id=None):
    children = []

    agent_count = len(agents)-1
    selected_agents_id = np.random.choice(top_parents_id,
agent_count)
    selected_agents = [agents[id] for id in selected_agents_id]
    child_agents = [mutate(agent) for agent in selected_agents]

    child_agents.append(elite(agents, top_parents_id, env))
    elite_id = len(child_agents)-1
    return child_agents, elite_id
```

8. Get the top parents:

```
>>def top_parents(scores, num_top_parents):
    return np.argsort(rewards)[::-1][:num_top_parents]
```

Here, we defined the function for identifying the elite agent and the function for adding noise to agents.

# How it works...

In this recipe, we looked at four different functions—the `mutation` function creates a duplicate for an agent and for each parameter, it appends a small random value that is limited by `MUTATION_POWER`. The `rand_like` method returns a tensor with the random values from a uniform distribution on the interval `[0, 1)` with the same size as `param`. Finally, the function returns the mutated child agent. Next, we saw that the `elite` function returns a copy of the best agent among the top performing agents. In the `elite` function, we reevaluate the agents to ensure that the agent with the highest score is picked as the elite and passed on as a child agent to the next generation.

The `child_agent` function generates child agents equal in number to the previous generation, where one of the children is an elite agent from the `elite` function and the rest are randomly chosen using `np.random.choice`. `selected_agents` holds the list of top-performing selected agents. In the `[mutate(agent) for agent in selected_agents]` step, the top-scoring agents are mutated using the `mutation` function.

Then, we appended the elite agent to the next generation of agents. Lastly, the `top_parent` function returns the indices of the top-performing agents in a generation.

# Training Deep GA

In this recipe, we will complete the evolution of the deep genetic algorithm and visualize the cartpole performing the balancing act. We will use all of the functions we've learned about in the recipes of this chapter and run for them for a given number of generations. This will create agents, get their scores, pick the best-performing agents, and mutate them for the next generation. Over the generations, we will see the score increase for the agents.

# How to do it...

Follow these steps:

1. First, we will import `gym`:

```
>>import gym
```

2. Next, we will declare the hyperparameters:

```
>>ENV_NAME = "CartPole-v1"
>>MAX_STEP = 500
>>MUTATION_POWER = 0.02
>>num_agents = 500
>>num_top_parents = 20
>>generations = 25
>>elite_agent = None
```

3. After that, we will create the environment and disable gradient calculations:

```
>>torch.set_grad_enabled(False)
>>env = gym.make(ENV_NAME)
```

4. Now, create the agents:

```
>>agents = create_agents(num_agents,
env.observation_space.shape[0], env.action_space.n)
```

5. Next, iterate over the generations:

```
>>print(f'| Generation | Score |')
>>for gen in range(generations):
```

Now, we can evaluate the agents:

```
        rewards = all_agent_score(agents, env, 3)
```

By doing this, we get the best agents:

```
        top_parents_id = top_parents(rewards, num_top_parents)
```

This, in turn, will create the next generation:

```
        agents, elite_agent = child_agents(agents, top_parents_id,
env, elite_agent)
        print(f'| {gen+1:03} | {np.mean([rewards[i] for i in
top_parents_id[:5]]):.4f} |')
```

The following code block shows a sample output:

```
| Generation |     Score       |
|    001     |    47.0667      |
|    002     |    47.3333      |
|    003     |    55.7333      |
|    004     |    58.2667      |
|    005     |    65.3333      |
|    006     |    88.0000      |
|    007     |    105.5333     |
|    008     |    117.4000     |
|    009     |    109.4000     |
|    010     |    137.6667     |
|    011     |    150.3333     |
|    012     |    168.6000     |
|    013     |    176.2667     |
|    014     |    248.0667     |
|    015     |    281.6667     |
|    016     |    327.9333     |
|    017     |    363.5333     |
|    018     |    375.4000     |
|    019     |    387.0000     |
|    020     |    432.2000     |
|    021     |    454.6000     |
```

```
|    022     |    445.9333     |
|    023     |    463.7333     |
|    024     |    482.1333     |
|    025     |    496.2000     |
```

6. Finally, we will visualize the cartpole's performance:

```
>>def play_agent(agent, env):
    observation = env.reset()
    total_reward=0
    for _ in range(MAX_STEP):
        env.render()
        observation =
torch.tensor(observation).type('torch.FloatTensor').view(1,-1)
        output_probabilities =
agent(observation).detach().numpy()[0]
        action = np.random.choice(range(2), 1,
p=output_probabilities).item()
        new_observation, reward, done, _ = env.step(action)
        total_reward += reward
        observation = new_observation

        if(done):
            break

    env.close()
    print("Rewards: ",total_reward)

>>play_agent(agents[num_agents-1],env)
Rewards: 350.0
```

With that, we have finished training and visualizing our DGA.

# How it works...

In this recipe, we evolved our deep genetic algorithm. We started by setting our hyperparameters and set `MUTATION_POWER` to `0.02`, as per the paper, *Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning*. We disabled gradient calculation using PyTorch since we don't need to rely on gradient descent to improve our model and create our environment.

Then, we created some agents in order to start our evolution and ran them for a predefined number of generations by iterating over `generations`, where we obtain the rewards of all the agents in each generation. Then, we picked the top-scoring parents and passed those parent indices to obtain the next generation using the `child_agent` function. After that, we printed out the mean of the top 5 scores.

Finally, we used the same `play_agent` function we used in the *Training DQN* recipe, with a minor modification, to compensate for the difference in the predicted value by the model. Here, we used the elite model to show the cartpole's performance, which, after each generation. sits at the end of the agents list. This was done using the `play_agent` function.

# There's more...

You can control the various hyperparameters of the deep genetic algorithm to see the differences in performance and store the scores to plot graphs.

# See also

You can read more about DGA at `https://arxiv.org/pdf/1712.06567.pdf` and `https://github.com/uber-research/deep-neuroevolution`.

# 8
# Productionizing AI Models in PyTorch

In this chapter, we will learn how to serve PyTorch model predictions to real-world problems. PyTorch has matured from a research tool to a production-ready framework and in this chapter, we will explore some of the features that allow PyTorch to be production-ready. Deploying a model means making your models available to your end users or systems. To do this, you might need to fulfill multiple requirements, such as being able to access the predictions over the web, making predictions fast for lower latency, or ensuring interoperability with other deep learning frameworks so that the developers can use the right tools as the project evolves. All of this ensures a faster transition from research to production.

In this chapter, we will cover the following recipes:

- Deploying models using Flask
- Creating a TorchScript
- Exporting to ONNX

## Technical requirements

All the recipes of this chapter have been completed using PyTorch 1.3 in Python 3.6.

# Deploying models using Flask

In this recipe, we will deploy an image classifier using the Flask microframework. The reason we're using Flask is because it's an easy-to-use microframework that can be used to build RESTful microservices, it is a very popular framework, and it is well documented. We will deploy an image classifier model that's been built using the Densenet-161 pre-trained model to complete this recipe.

# Getting ready

We will need to install Flask for this recipe. Use the `pip` manager to install `flask`:

```
pip install flask
```

With this, we can get started.

# How to do it...

We will divide this recipe into multiple files. Follow these steps to do so:

1. Create a file called `image_classifier.py`.
2. Now, we need to make our imports:

```
>>import io
>>import torch
>>from torchvision import models
>>from PIL import Image
>>import torchvision.transforms as transforms
>>import json
```

3. Read the `.json` file containing the class names:

```
>>with open('idx_class.json') as f:
    idx_class = json.load(f)
```

4. Define the `create_model` function:

```
>>def create_model():

    model_path = "densenet161.pth"
    model = models.densenet161(pretrained=True)
    model.load_state_dict(torch.load(model_path,
map_location='cpu'), strict=False)
```

```
    model.eval()
    return model
```

5. Define the `image_transformer` function:

```
>>def image_transformer(image_data):
    transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std=[0.229, 0.224, 0.225])
    ])

    image = Image.open(io.BytesIO(image_data))
    return transform(image).unsqueeze(0)
```

6. Next, we need to define the `predict_image` function:

```
>>def predict_image(model, image_data):

    image_tensor = image_transformer(image_data)
    output = model(image_tensor)
    _, prediction = output.max(1)
    object_index = prediction.item()

    return idx_class[object_index]
```

7. Now, we will create `imageapp.py`.
8. First, we will import the required modules and Flask:

```
>>from flask import Flask, request, jsonify
>>from image_classifier import create_model, predict_image
```

9. Now, we will create a Flask app and the classifier model:

```
>>app = Flask(__name__)
>>model = create_model()
```

10. Now, let's create the route:

```
>>@app.route('/predict', methods=['POST'])
```

11. Next, we will write a function that will be invoked on this route:

```
>>@app.route('/predict', methods=['POST'])
>>def predicted():
      if 'image' not in request.files:
          return jsonify({'error': 'Image not found'}), 400

      image = request.files['image'].read()
      object_name = predict_image(model, image)
      return jsonify({'object_name' : object_name})
```

12. Finally, we will start the Flask app if `imageapp.py` is run:

```
>>if __name__ == '__main__':
    app.run(debug=True)
```

13. Next, you need to run the Flask app using the following command:

```
python imageapp.py
```

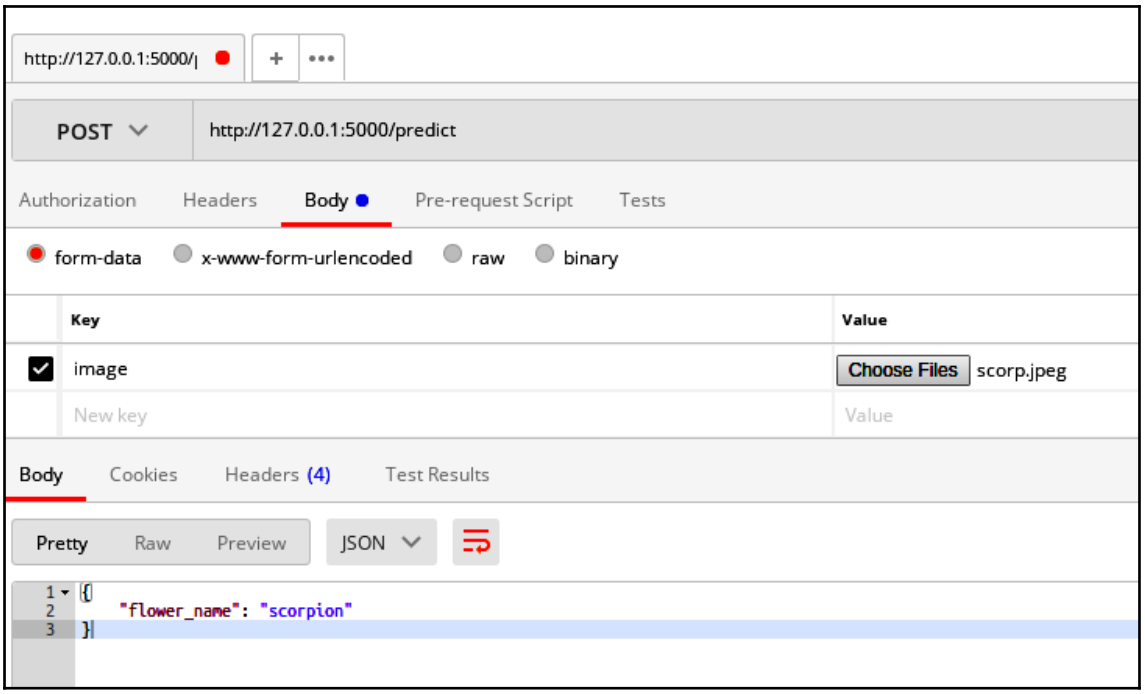
    By running this command, the Flask server will be up and running. You should be able to access the app's URL at `http://127.0.0.1:5000/` and send a `POST` request.

14. Using the Postman tool, you can check the API. Here is a sample response for the image:

    Now, we will check the API's response:

```
{
 "object_name": "scorpion"
 }
```

This will give us the following output:




In this recipe, we did a simple app deployment with Flask.

# How it works...

In this recipe, we deployed a model for inference using RESTful APIs using the Flask Python framework. We started by creating `image_classifier.py` and loaded the class names from `idx_class.json`.

In this file, the first function loads a pre-trained `densenet161` model that is trained on the ImageNet dataset with 1,000 classes; we set the model in evaluation mode and returned the model. The second function converts a given input image into a tensor and applies transforms on it. We used the `Image` module in `PIL` to read the image data. The third function makes the prediction on a given image by converting it into a tensor and passing it into the model. This returns the name of the object in the image.

Then, we switched over to the `imageapp.py` file, where we used Flask to create the web app. Here, we created the Flask app using `app = Flask(__name__)` and created the model using the `create_model` function. After this, we created a route called `/predict`, which receives a `POST` request using the app instance we created. Then, we defined the `predicted` function, which will be invoked upon calling the `/predict` URL. `request.files` holds the files in the POST requests. Here, we checked whether the image file is uploaded using the post parameter name, that is, `image`.

Finally, we passed this image data into the `predict_image` function we defined earlier. The `jsonify` method in Flask ensures the response is in `.json` format. `app.run(debug=True)` starts the Flask server and serves the request.

# There's more...

In this recipe, we set debug mode to on using `debug=True`, which is not recommended in production. The Flask server is not powerful enough to support production loads. Instead, you should use `gunicorn` and `nginx` for proper deployment.

# See also

You can read more about Flask at: `http://flask.palletsprojects.com/en/1.1.x/`.

You can learn about gunicorn Nginx deployment at: `https://www.digitalocean.com/community/tutorials/how-to-serve-flask-applications-with-gunicorn-and-nginx-on-ubuntu-18-04`.

You can see an alternate implementation at: `https://pytorch.org/tutorials/intermediate/flask_rest_api_tutorial.html`.

# Creating a TorchScript

TorchScript provides an intermediate representation for models that were originally written in PyTorch. With this, you can run the models in a high-performance environment such as C++. TorchScript creates serializable and optimizable versions of models from PyTorch code. Code written in TorchScript can be loaded into a process without any Python dependencies. TorchScript provides tools that can be used to capture the model definition, and PyTorch is capable enough to support this, thanks to its dynamic and flexible nature. It is possible to create a TorchScript in two ways: tracing, or using a script compiler. In this recipe, we will convert a PyTorch model into TorchScript using tracing and a script compiler.

# How to do it...

In this recipe, we will create a TorchScript. Follow these steps to do so:

1. First, we will write a simple network:

```
>>import torch
>>import torch.nn as nn
>>class MyCell(torch.nn.Module):
      def __init__(self):
          super(MyCell, self).__init__()
          self.linear = torch.nn.Linear(4, 4)

      def forward(self, x, h):
          new_h = torch.tanh(self.linear(x) + h)
          return new_h
```

2. Next, we will create a model from the model class:

```
>>my_cell = MyCell()
```

3. Then, we will generate two random tensors to be passed into the model:

```
>>x, h = torch.rand(4, 4), torch.rand(4, 4)
```

4. Next, we can `jit.trace`:

```
>>traced_cell = torch.jit.trace(my_cell, (x, h))
>>traced_cell


TracedModule[MyCell](
  original_name=MyCell
```

```
  (linear): TracedModule[Linear](original_name=Linear)
)
```

5. Then, we pass the tensors to `traced_cell`:

```
>>traced_cell(x, h)


tensor([[ 0.4238, -0.0524, 0.5719, 0.4747],
        [-0.0059, -0.3625, 0.2658, 0.7130],
        [ 0.4532, 0.6390, 0.6385, 0.6584]],
       grad_fn=<DifferentiableGraphBackward>)
```

6. We can access the graphs using the following code:

```
>>traced_cell.graph


graph(%self : ClassType<MyCell>,
      %input : Float(3, 4),
      %h : Float(3, 4)):
  %1 : ClassType<Linear> = prim::GetAttr[name="linear"](%self)
  %weight : Tensor = prim::GetAttr[name="weight"](%1)
  %bias : Tensor = prim::GetAttr[name="bias"](%1)
  %6 : Float(4, 4) = aten::t(%weight), scope: MyCell/Linear[linear]
# /home/<user>/.local/lib/python3.6/site-
packages/torch/nn/functional.py:1370:0
  %7 : int = prim::Constant[value=1](), scope:
MyCell/Linear[linear] # /home/<user>/.local/lib/python3.6/site-
packages/torch/nn/functional.py:1370:0
  %8 : int = prim::Constant[value=1](), scope:
MyCell/Linear[linear] # /home/<user>/.local/lib/python3.6/site-
packages/torch/nn/functional.py:1370:0
  %9 : Float(3, 4) = aten::addmm(%bias, %input, %6, %7, %8), scope:
MyCell/Linear[linear] # /home/<user>/.local/lib/python3.6/site-
packages/torch/nn/functional.py:1370:0
  %10 : int = prim::Constant[value=1](), scope: MyCell # <ipython-
input-2-c6e2cd8665ee>:7:0
  %11 : Float(3, 4) = aten::add(%9, %h, %10), scope: MyCell #
<ipython-input-2-c6e2cd8665ee>:7:0
  %12 : Float(3, 4) = aten::tanh(%11), scope: MyCell # <ipython-
input-2-c6e2cd8665ee>:7:0
  return (%12)
```

For a readable version of this, we can use the following command:

```
>>traced_cell.code


import __torch__
import __torch__.torch.nn.modules.linear
def forward(self,
            input: Tensor,
            h: Tensor) -> Tensor:
    _0 = self.linear
    weight = _0.weight
    bias = _0.bias
    _1 = torch.addmm(bias, input, torch.t(weight), beta=1, alpha=1)
    return torch.tanh(torch.add(_1, h, alpha=1))
```

Now, let's explore the script compiler. Follow these steps:

1. First, we will define a submodule with control flow:

```
>>class MyDecisionGate(torch.nn.Module):
    def forward(self, x):
        if x.sum() > 0:
            return x
        else:
            return -x
```

2. Then, we will use this submodule in the model definition:

```
>>class MyCell(torch.nn.Module):
    def __init__(self, dg):
        super(MyCell, self).__init__()
        self.dg = dg
        self.linear = torch.nn.Linear(4, 4)

    def forward(self, x, h):
        new_h = torch.tanh(self.dg(self.linear(x)) + h)
        return new_h
```

3. Create a model from the definition:

```
>>my_cell = MyCell(MyDecisionGate())
```

4. Now, we will perform the trace:

```
>>traced_cell = torch.jit.trace(my_cell, (x, h))
>>traced_cell.code
```

```
import __torch__.___torch_mangle_0
import __torch__
import __torch__.torch.nn.modules.linear.___torch_mangle_1
def forward(self,
    input: Tensor,
    h: Tensor) -> Tensor:
  _0 = self.linear
  weight = _0.weight
  bias = _0.bias
  x = torch.addmm(bias, input, torch.t(weight), beta=1, alpha=1)
  _1 = torch.tanh(torch.add(torch.neg(x), h, alpha=1))
  return _1
```

Next, we will convert this into TorchScript with `jit.script`:

```
>>scripted_gate = torch.jit.script(MyDecisionGate())
>>my_cell = MyCell(scripted_gate)
>>traced_cell = torch.jit.script(my_cell)
>>print(traced_cell.code)
```

```
import __torch__.___torch_mangle_3
import __torch__.___torch_mangle_2
import __torch__.torch.nn.modules.linear.___torch_mangle_4
def forward(self,
    x: Tensor,
    h: Tensor) -> Tensor:
  _0 = self.linear
  _1 = _0.weight
  _2 = _0.bias
  if torch.eq(torch.dim(x), 2):
    _3 = torch.__isnot__(_2, None)
  else:
    _3 = False
  if _3:
    bias = ops.prim.unchecked_unwrap_optional(_2)
    ret = torch.addmm(bias, x, torch.t(_1), beta=1, alpha=1)
  else:
    output = torch.matmul(x, torch.t(_1))
    if torch.__isnot__(_2, None):
      bias0 = ops.prim.unchecked_unwrap_optional(_2)
      output0 = torch.add_(output, bias0, alpha=1)
    else:
      output0 = output
    ret = output0
  _4 = torch.gt(torch.sum(ret, dtype=None), 0)
  if bool(_4):
    _5 = ret
```

```
  else:
    _5 = torch.neg(ret)
  return torch.tanh(torch.add(_5, h, alpha=1))
```

With this, we have looked at two different ways to create a TorchScript.

# How it works...

In this recipe, we used the tracing method to create a TorchScript. We defined a simple module called `MyCell` to be converted into `Torchscript` and created two sample tensors called `x` and `h` to be passed into the forward method of the network module. Then, we used `jit.trace` to trace the Python code and create the TorchScript.

We converted a `PyTorch` model into TorchScript using tracing and passed the instance of our model. `jit.trace` creates a `torch.jit.ScriptModule` object by tracing the operations in our model evaluation within the module's forward method. `jit.trace` runs the network module, records the operations that occurred when the module was run, and creates an instance of the `torch.jit.ScriptModule` object. TorchScript records its definitions in an intermediate representation (referred to as a graph in deep learning). Then, we examined the graph with the `.graph` property and generated a more readable version using `.code`, which is the Python syntax interpretation of the code.

Then, we explored the next method of creating a TorchScript, which was by using a script compiler. For this, we defined a submodule with a control flow using the following code:

```
>>class MyDecisionGate(torch.nn.Module):
    def forward(self, x):
        if x.sum() > 0:
            return x
        else:
            return -x
```

We used the following submodule in our `MyCell` module:

```
my_cell = MyCell(MyDecisionGate())
```

With the tracing method, we lose the flow of control since, with tracing, we ran the code, recorded the operations, and constructed a `ScriptModule` object that erases things such as control flow. This can be seen in the following code:

```
>>traced_cell = torch.jit.trace(my_cell, (x, h))
>>traced_cell.code
```

Due to this, we use `jit.script`, which preserves the control flow. First, we run `jit.script` on the submodule object, as follows:

```
>>scripted_gate = torch.jit.script(MyDecisionGate())
```

Then, we create the `MyCell` object and run it with `jit.script`:

```
>>my_cell = MyCell(scripted_gate)
>>traced_cell = torch.jit.script(my_cell)
```

When we printed the TorchScript code using `print(traced_cell.code)`, we saw that the flow of control was still preserved.

## There's more...

We can mix the tracing and scripting methods together.

## See also

You can find out more about mixing tracing and scripting at: `https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html#mixing-scripting-and-tracing`.

# Exporting to ONNX

In this recipe, we will look at exporting PyTorch models into the **Open Neural Network Exchange** (**ONNX**), which provides an open source format for both deep learning and traditional machine learning models. It defines an extensible computation graph model, as well as built-in operators and standard data types.

ONNX is widely supported and can be found in many frameworks, tools, and hardware since it enables interoperability between different frameworks and transitioning from research to production.

# Getting ready

For this recipe, we need to install ONNX, which we can install using the following command:

```
pip install onnx
```

With this, we can proceed with the recipe.

We will also need the trained weights of the model we trained on `CIFAR-10` in `Chapter 3`, *Convolutional Neural Networks for Computer Vision*, for this recipe.

# How to do it...

In this recipe, we will export our CIFAR-1o model into ONNX format and run it using `onnxruntime`. Follow these steps to do so:

1. We will start with the imports:

```
>>import onnx
>>import onnxruntime
>>import torch.nn as nn
>>import torch
>>import torch.nn.functional as F
>>import numpy as np
```

2. Next, we will define the model class:

```
>>class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.linear1 = nn.Linear(64 * 4 * 4, 512)
        self.linear2 = nn.Linear(512, 10)
        self.dropout = nn.Dropout(p=0.3)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 64 * 4 * 4)
        x = self.dropout(x)
        x = F.relu(self.linear1(x))
        x = self.dropout(x)
```

```
        x = self.linear2(x)
        return x
```

3. Then, we will create the model object and load the weights from our training:

```
>>model = CNN()
>>model.load_state_dict(torch.load("cifar10.pth"))
<All keys matched successfully>
```

4. Next, we will set the model in evaluation mode:

```
>>model.eval()

CNN(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (linear1): Linear(in_features=1024, out_features=512, bias=True)
  (linear2): Linear(in_features=512, out_features=10, bias=True)
  (dropout): Dropout(p=0.3, inplace=False)
)
```

5. Now, we will create a random variable:

```
>>x = torch.randn(1, 3, 32, 32, requires_grad=True)
```

6. Next, we will get the output for the random variable, `x`:

```
>>model_out = model(x)
```

7. After that, we will export the model and its weight into the `onnx` model:

```
>torch.onnx.export(model,
                   x,
                   "cifar.onnx",
                   export_params=True,
                   opset_version=10,
                   do_constant_folding=True,
                   input_names = ['input'],
                   output_names = ['output'],
                   dynamic_axes={'input' : {0 : 'batch_size'},
                                 'output' : {0 : 'batch_size'}})
```

8. Next, we will load and check the `onnx` model:

```
>>onnx_model = onnx.load("cifar.onnx")
>>onnx.checker.check_model(onnx_model)
```

9. We will load `onnx` into the ONNX runtime:

```
>>ort_session = onnxruntime.InferenceSession("cifar.onnx")
```

10. Now, define the `to_numpy()` function:

```
>>def to_numpy(tensor):
    return tensor.detach().cpu().numpy() if tensor.requires_grad
else tensor.cpu().numpy()
```

11. Here, we will pass the input variable, `x`, into the ONNX runtime:

```
>>ort_inputs = {ort_session.get_inputs()[0].name: to_numpy(x)}
>>ort_outs = ort_session.run(None, ort_inputs)
```

12. Finally, we will check if the outputs from the model and the `onnx` model are equal:

```
>>np.testing.assert_allclose(to_numpy(model_out), ort_outs[0],
rtol=1e-03, atol=1e-05)
```

With this recipe, we have exported to `onnx` format and ran a model in `onnx` format using the ONNX runtime.

# How it works...

In this recipe, we exported a normal PyTorch model into ONNX format and ran the `onnx` model using the ONNX runtime. For this, we took a model with a weight. Here, we used the CIFAR-10 model from `Chapter 3`, *Convolutional Neural Networks for Computer Vision*. We used the weights of the model from the training and set the model in evaluation mode for fast and light computation.

Then, we used a random variable with the same shape as that of an input tensor, which in our case is a three-channel 32 x 32 pixel image. We pass this random input into our model and obtain the output. Then, we used the output to compare it with the model from the ONNX version of the model.

Exporting a model happens in PyTorch using either tracing or scripting. In this recipe, we used tracing with the help of `torch.onnx.export()`. Tracing keeps track of the operations that are used to obtain the output. This is why we provided `x` – so that tracing is possible. `x` must have the right type and size. The input size is fixed in the exported ONNX graph for all the input's dimensions, and we must specify all the dynamic axes. In this recipe, we exported the model with an input of the first dimension, set the batch size to 1, and specified the first dimension as dynamic in the `dynamic_axes` parameter in `torch.onnx.export()`.

The first argument is the PyTorch model, while the second is the random variable. Then, we have the path for the `onnx` format; `export_params` is used to store the trained parameter weights inside the model file; `opset_version` is the `onnx` export version; `do_constant_folding` is used to execute constant folding for optimization; `input_names` is the model's input name, and `output_names` is the model output names. Then, we loaded the exported `onnx` model and checked the model structure and validated the schema using `onnx.checker.check_model(onnx_model)`. The ONNX graph is verified by checking the model version, the graph's structure, the nodes, and their inputs and outputs.

Then, we loaded the model in the `onnx` runtime and created an inference session for the model. Once the session was created, we evaluated the model using the `run()` API, where the first parameter is a list of output names and the second parameter is the input dictionary. The output of this call is a list of outputs from the model after computing the ONNX runtime. Finally, we compared the output values from the PyTorch model and the `onnx` model using `numpy.testing.assert_allclose()`, which raises an `AssertionError` if two objects are not equal up to desired tolerance.

# There's more...

We could export the `onnx` model, load in other supported frameworks, and configure the export using other parameters in `torch.onnx.export()`.

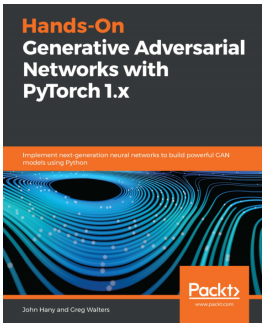
# See also

You can read more about ONNX at: `https://pytorch.org/tutorials/advanced/super_resolution_with_onnxruntime.html`.

You can read more about the Python ONNX runtime at: `https://microsoft.github.io/onnxruntime/python/index.html`.

# Other Books You May Enjoy

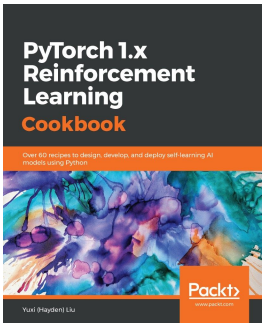
If you enjoyed this book, you may be interested in these other books by Packt:




**Hands-On Generative Adversarial Networks with PyTorch 1.x**
John Hany, Greg Walters

ISBN: 978-1-78953-051-3

- Implement PyTorch's latest features to ensure efficient model designing
- Get to grips with the working mechanisms of GAN models
- Perform style transfer between unpaired image collections with CycleGAN
- Build and train 3D-GANs to generate a point cloud of 3D objects
- Create a range of GAN models to perform various image synthesis operations
- Use SEGAN to suppress noise and improve the quality of speech audio




**PyTorch 1.x Reinforcement Learning Cookbook**
Yuxi (Hayden) Liu

ISBN: 978-1-83855-196-4

- Use Q-learning and the state–action–reward–state–action (SARSA) algorithm to solve various Gridworld problems
- Develop a multi-armed bandit algorithm to optimize display advertising
- Scale up learning and control processes using Deep Q-Networks
- Simulate Markov Decision Processes, OpenAI Gym environments, and other common control problems
- Select and build RL models, evaluate their performance, and optimize and deploy them
- Use policy gradient methods to solve continuous RL problems

# Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

# Index

# O



# P



# Q



# R



# S



# T

# V



# W



# Z